Avoiding Excessive Data Exposure through Microservice APIs

Patric Genfer[0000−0002−4236−5951] and Uwe Zdun[0000−0002−6233−2591]

Research Group Software Architecture, Faculty of Computer Science,
University of Vienna, Vienna, Austria
{patric.genfer|uwe.zdun}@univie.ac.at

Abstract. Data transfer and exchange of information through APIs are essential for each microservice architecture. Since these transfers often include private or sensitive data, potential data leaks, either accidentally or through malicious attacks, provide a high-security risk. While there are different techniques, like using data encryption or authentication protocols to secure the data exchange, only a few strategies are known to reduce the damage when an actual data breach happens. Our work presents a novel approach to identifying the optimal amount of data attributes that need to be exchanged between APIs and minimizes the damage in case of a potential breach. Our method relies only on static source code analysis and easy-to-calculate architectural metrics, making it well suited to be used in continuous integration and deployment processes. We further verified and validated the feasibility of our approach by conducting two case studies on open-source microservice systems.

Keywords: Microservice API · security · data exposure · metrics · source code detectors.

1 Introduction

Despite their main principles like autonomy and isolation, microservices often have to cooperate and interchange data to fulfill their tasks [1]. These interservice communications mainly consist of domain-specific information required to execute particular tasks that might include user-related or other sensitive data [2]. While business-related information can be of high interest for malicious competitors, it is especially the sensitive data that presents a lucrative target for external attackers intent on either stealing or compromising the information [3]. It is therefore not surprising that the exposure of sensitive data is considered an essential security challenge in microservices architectures [4].

Data exchange happens also in monolithic architecture, but here, the attack surface for data-related attacks is much smaller since a large part of the communication takes place inside the process boundaries and is therefore not as easily accessible from the outside [5]. In contrast, microservices communicate via different data transfer technologies [6] by using a variety of data formats [7], and their endpoints are often accessible through the public cloud, making data attacks much more likely and rewarding [3, 5].

To protect the data transfer against leaks or breaches, different security mechanisms have been proposed, such as authentication, authorization, traffic control, and encryption [3, 5, 8]. However, all these measures come with the price of having a considerable negative impact on other quality properties, such as the system's performance. Yarygina and Bagge [9], for instance, could show that their security framework affects the overall system performance by 11%. In addition, excessive data transfer between services can cause unintended concurrency issues, making it even harder to reason about a system's security aspects [6].

Since data autonomy and isolation are key characteristics of a microservice architecture [10], exchanging data between services is essential and avoiding data transfer at all is not an option for obvious reasons. Dias and Siriwardena [5] refer to data transfer that goes beyond what is strictly necessary, as *excessive data exposure* and further suggest that each API should only provide precisely that part of the information required by its consumers.

Unfortunately, identifying the amount of data mistakenly interchanged between services is not a trivial task: Many APIs were not designed with data parsimony as a primary goal, instead the focus during design is usually on quality goals such as improving maintainability and reducing complexity, resulting in a coarser-grained API structure than sometimes necessary [11]. Beside that, APIs based on underdeveloped or anemic domain models carry the risk of exposing too much and often domain data unrelated to specific use cases [12]. Finally, the polyglot and diverse nature of microservice systems [9] needs to be considered, making optimizing for data parsimony an even more challenging task.

Our work provides a novel approach for tracking the data transferred between microservice APIs and aims to identify any excessive data exposure that happens as part of this transfer[1]. To achieve this, we use our source code detector approach from our previous research [13] to derive a communication and data flow model from the underlying system's code artifacts. Based on this model, we define a set of architectural metrics for guiding architectural design decisions targeting the reduction of excessive data exposure. In this context, we are aiming at answering the following research questions:

RQ1 *How can a communication model for identifying excessive data exposure be derived from a microservice system?* Our goal is to reconstruct such a model only through static code analysis, making our approach exceedingly feasible for continuous development cycles.

RQ2 *How well can the level of data exposure caused by API calls be quantified?* Based on our formal model, we will identify a set of architectural metrics to measure and identify the grade of data exposure through an API.

RQ3 *How can software architects be guided through the process of redesigning microservice APIs to reduce excessive data exposure?* We will investigate how our metrics can help by restructuring critical architecture elements to reduce the amount of data exposed.

[1] For supporting reproducability, we offer the whole source code and data of our study in a data set published on the long term archive Zenodo: `https://zenodo.org/record/6700021#.YrRJYHVByA0`.

This paper is structured as follows: Section 2 gives a short overview of existing research in this area. Section 3 presents some background information regarding the data management in microservice APIs and how it could lead to excessive data exposure. Section 4 describes our communication model and how we constructed it by using our source code detectors. Section 5 introduces the metrics we designed to measure potential data exposure, and in Section 6, we assess our metrics on two open-source case studies. The paper concludes with a discussion of our results (Section 7), together with an overview of possible threats to validity and a selection of future work tasks (Section 8 and 9).

2 Related Work

Microservice security, especially data exposure, is an important research field that has attracted much attention in recent years. Yu et al. [3] investigate security issues in microservice-based fog applications. Similar to our research, they consider inter-service communication a critical security aspect. A more data-centric security approach is pursued by Miller et al. [2], where the authors argue that leaking data can pose an enormous financial risk for companies and accordingly present a security architecture to prevent these data exposures while still enforcing the required business workflows. Shu et al. [14] introduce a method to detect sensitive data exposure in microservice systems by preserving data privacy. For more research in this area, see also Hannousse and Yahiouche [4].

Another essential data security concept is the so-called *taint analysis* that examines how untrusted data – e.g., from user input – can affect the security of a system, be it a web [15] or mobile application [16]. While similar to our approach, taint analysis is a vast research field that focuses on how untrusted data could potentially harm a system. In contrast, our approach is more specific as it aims to minimize the amount of data exposed through microservice APIs.

When it comes to gathering data from microservices through mining source code or other artifacts, Soldani et al. [17] present an approach in which they reconstruct a communication model by parsing Kubernetes configuration files. While their model is similar to ours, they do not analyze the source code directly and therefore have limitations regarding the actual data usage by an API. Fowkes and Sutton [18] introduce an approach for mining API call patterns by creating abstract syntax trees (AST) out of Java programs to reconstruct method invocations. In contrast, our code detector approach is more lightweight and not specific to a particular language.

Studies about architectural metrics related to microservice can be found in [19] and [20]. Although their metrics are not specific for measuring data exposure, they provide meaningful insights we also used to define our metrics.

3 Excessive Data Exposure in Microservice APIs

To fulfill their tasks, microservices have to manage and process a large amount of data and information, provided by external sources like databases or computed or generated during runtime. A typical pattern here is to provide each microservice with its own database [6], placed inside the internal perimeter of the service boundary. Because of their central role in data-driven architectures, we assume that for each database a data provenance protection exists and they are secured against data breaches [3]. Otherwise, an attacker could target the database directly instead of focusing on the communication channels.

In addition, a microservice API itself can act as a data provider, either by computing derived data from incoming parameters [21] or by transferring data directly from one API to another. To decide whether an API connected to a data source requests its data legitimately, we have to investigate the following cases:

1. *The API processes the incoming data directly.* This would be the case if the API either stores its input in its data store or uses it to derive new information. Consider e.g. an online shop system API that calculates a discount factor based on the customer's shopping history it receives. Since the incoming data is no longer required after being processed, we say the API *consumes* the input data.
2. *The API sends incoming data to another target without modification.* Besides directly consuming incoming information, an API can also leave the input untouched and transfer it directly to another API, acting itself as a data source for its caller. We consider this behavior as *routing*.

If any incoming data is not handled by one of these two mechanisms, we can assume that the API's input is obviously not required and can safely – at least from a security point of view – be omitted. This leads us to the following definition:

Definition 1. *All incoming data received by an API that is neither directly consumed nor routed to another API can be considered* ***excessively exposed****.*

Figure 1 illustrates this definition: An external client communicates with a microservice system consisting of three APIs, requesting order information by calling the `/order/collect-data` operation. The API itself gathers the required data by calling two other endpoints, `/customer` and `/product`. Both APIs return complex objects containing the required data attributes and additional information not relevant for the current use case (marked with red color). While `/order/collect-data` returns only the required information to the client, it has no use for the remaining data attributes `movementHistory, description` and `image`, which, while not used by any endpoints, are still exposed to the network. If the API communication had been compromised by an attacker, they would get access to much more data than one would actually expect based on the underlying use case, making the data breach even more severe.

How could we avoid this unnecessary and excessive exposure of data? First, we have to determine the original source of the data by following the path back from our current API to the one that initially returned the data attributes. Secondly, after investigating our two origin APIs, we recognize that they are not tailored for a specific use case but instead simply return whole data entities, thus exposing too many details [12].

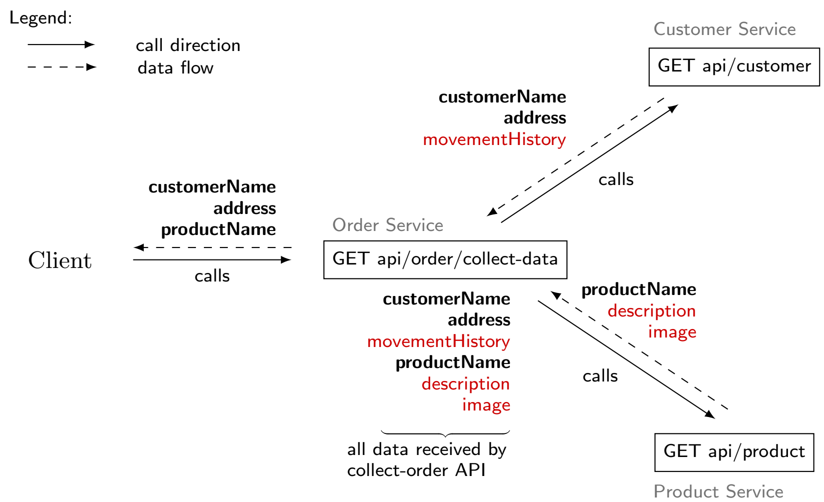

Fig. 1: Example of Excessive Data Exposure: Two APIs are exposing more data than actually required for the use case.

A naive solution would split up the `api/customer` and `api/product` operations into separate APIs for each single data attribute, allowing clients a more fine-grained access to choose each data attribute individually. While this would minimize the risk of a potential data leak, the solution comes with many disadvantages: A single API endpoint per data attribute increases the size of the service interface significantly and also the efforts necessary to implement and maintain the service [19]. Also, a client in need of more than one data attribute at once has to make several network calls to gather all required information, resulting in increased network traffic and poorer system performance.

Somewhere between these two extremes – one endpoint per atomic data attribute vs. one endpoint per domain entity – lies the optimal solution from a data security point of view: Designing the APIs per use case and ensuring that each request receives only the data relevant for the case it implements. In our example, we could design one API `api/customer/order-information` returning the `name` and the `address` attributes and another one returning only the name of the product (see Figure 2 for an illustration of these three options).

However, even such an optimized solution has drawbacks, as systems with a large number of different use cases would require a corresponding amount of API endpoints, resulting again in increased implementation and maintenance efforts. Furthermore, providing one API per use case creates a strong coupling between the both, reducing the maintainability even further [12]. Choosing the right granularity for a service interface is, therefore, often a decision that depends on many different individual factors [11]. Unfortunately, architects are often left alone in this process and have to rely on their experience or personal preferences to find the optimal API granularity. Our approach presented in the next section closes this gap and supports experts by providing them with additional guidance when redesigning their architectures towards more data security.

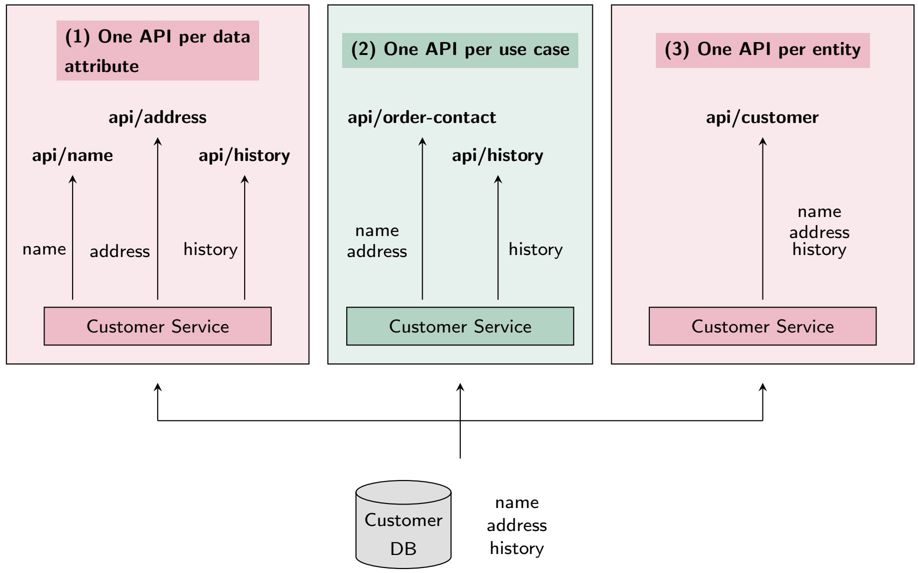

Fig. 2: Different types of API granularity depending on the amount of data exposed per API.

4 Communication Model

4.1 Formal Model Description

The communication model we extract from the underlying microservice artifacts is derived from our previous work [13]. We express inter-service calls within the system as a directed graph $G = (V, E, DA, F)$, where V represents the set of architectural elements essential for the communication flow – like APIs ($V^{API} \in V$) – and $E \subseteq V \times V$ is the set of invocations (each represented by a tuple), both synchronous and asynchronous. We further define DA as the set of all data attributes interchanged and processed by the APIs $v \in V^{API}$ of a Graph G. The last part of our definition specifies F as a set of data-related functions with $F = \{in, out, consume\}$ and $in : V^{API} \to \mathbb{P}(DA)$ (same for out and $consume$). Each of these functions returns the subset of data attributes $\mathbb{P}(DA)$ an API either receives as input or processes.

4.2 Source Code Mining

Code artifacts of a microservice system are often written in various languages and use a large number of different communication and database technologies [6]. Strategies for mining these artifacts with native language parsers would require considerable configuration and maintenance work and are therefore not always practical, or often limited to specific languages (see, for instance [18] or [22]).

Instead, we used a different mining approach adapted from our previous works [13, 23]. for mining code artifacts: To identify significant architectural hotspots within a system, it is often sufficient to look only for specific patterns in code that describe a particular communication model or API technology and ignore all other parts that are not relevant in this context. Taking advantage of this fact allows us to implement much more lightweight parsers – we call them detectors – that focus only for detecting specific patterns [23].

In this paper, we substantially extended our detector concept, to enable a more reusable and lightweight detection approach. In particular, we reduced the size of each detector by splitting up responsibilities further through introducing the concept of *Collectors*: While a *Detector* is responsible for identifying an architectural hotspot based on characteristics it finds in an artifact, a Collector's job is to extract relevant information – like an API endpoint or a REST method parameter – from the previously identified artifact. This approach resulted in a multi-phase parsing strategy, where first relevant code artifacts are localized by detectors, and then different collectors run over the artifacts to extract all relevant information.

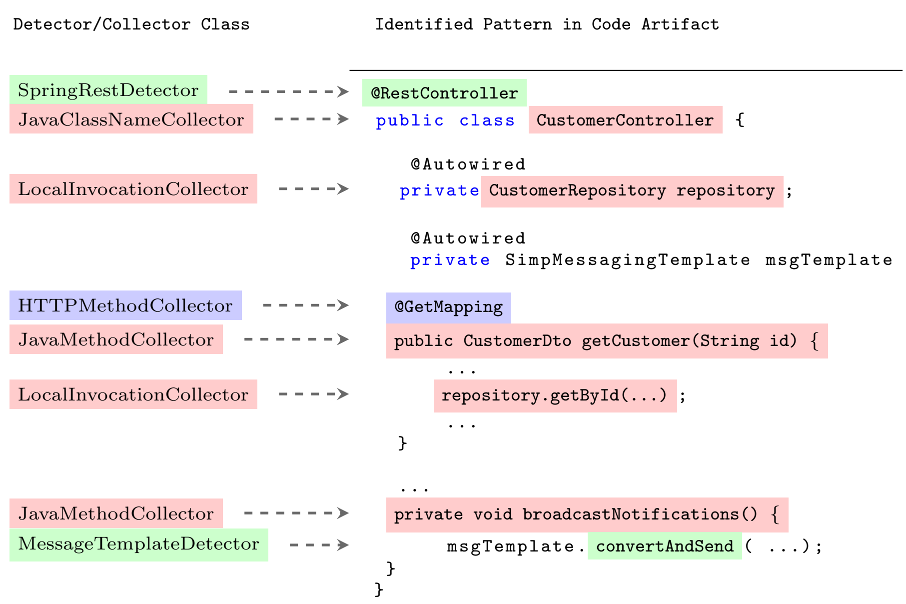

Fig. 3: Source Code Mining through *Detectors* and *Collectors*: Detectors (*Green*) identify relevant architectural hotspots. Language-specific Collectors (*Red*) extract semantic information depending on the language used. Other Collectors (*Blue*) search for specific technology-related patterns.

Figure 3 illustrates the roles of some example *Detectors* and *Collectors* during processing of a source code file artifact. As seen, a single hotspot can have more than one architectural role. Here, the class `CustomerController` provides different synchronous API endpoints, but also propagates asynchronous messages.

After collecting all hotspots, our algorithm creates the communication model by following local and inter-service invocations and storing the incoming and outgoing API parameters for later processing.

5 Metrics

According to our definition from Section 3, an API $v \in V^{API}$ exposes data excessively if not all incoming data attributes are either consumed by or routed to the API's output. On the basis of this observation, we define our main metric $EDE_{API} : V^{API} \rightarrow \mathbb{P}(DA)$ (*E*xcessive *D*ata *E*xposure) as the relative complement of the sets of incoming data attributes $in(v)$ versus the set of all processed data attributes ($out(v)$ and $consume(v)$):

$$EDE_{API}(v) = in(v) \setminus \Big(out(v) \cup consume(v)\Big) \quad with\, v \in V^{API} \tag{1}$$

Since this metric returns a set of data attributes, a more convenient way for using it in production environments, such as CI pipelines or quality dashboards, would be to have a single indicator $isEDE_{API} : V^{API} \rightarrow Boolean$ instead:

$$isEDE_{API}(v) = \begin{cases} true, & \text{if } EDE_{API}(v) \neq \varnothing \quad with\, v \in V^{API} \\ false, & \text{otherwise} \end{cases} \tag{2}$$

While all types of excessive data exposure can have adverse effects on a system's performance because of an increased message payload, not all of them necessarily impact the security of a system in the same way. But the more sensitive the data is, the more severe the damage is in case of a potential data leak. To provide a way of measuring the severity of such a leak, we introduce an ordinal scale with three risk categories that are relatively common among various public organizations[2]. The three categories are:

- **Low**: Data of this category is publicly available, such as a person's contact information or other data shared through a public profile.
- **Moderate**: This category covers data that is not publicly available or might fall under the GDPR privacy regulations.
- **High**: This information is considered the most sensitive and includes all kinds of private data, especially IDs such as social or credit card numbers and health or financial records.

Of course, whether a data record is considered sensitive or not tends to be a highly subjective decision, as the privacy of a data attribute often depends on the use case and whether the data-leaking systems are publicly accessible [24]. Since even a single high-risk data leak is a critical incident, a metric to measure the severity of an API's data exposure has to take this into account. To define our *Excessive Data Exposure Severity* $EDES_{API} : V^{API} \rightarrow SL$ metric, we, therefore, take the data attribute with the highest severity level to determine the overall severity:

$$EDES_{API}(v) = \max\Big(\{s \in SL : s = severity(d) \wedge d \in EDE_{API}(v)\}\Big) \tag{3}$$

With $SL = \{Low, Moderate, High\}$ being a set of severity levels and $severity(d) : DA \rightarrow SL$ a function assigning each data attribute one of the three severity values. This mapping has to be done upfront by a domain expert.

The easiest way to reduce data exposure would be to remove all excessively exposed data attributes from the system, but this is only possible if the specific

[2] `https://uit.stanford.edu/guide/riskclassifications#risk-classifications`

data attribute is not used in another part of the implementation. To identify such 'orphaned' data, we can use the following *Usage Exposure Ratio* metric $UER_{data} : DA \to \mathbb{R}$ for a given data attribute $d \in DA$:

$$UER_{data}(d) = \frac{|\{v \in V^{API} : d \in consume(v)\}|}{|\{v \in V^{API} : d \in EDE_{API}(v)\}|} \tag{4}$$

Values larger than 0 indicate that at least one API is consuming a data attribute, and therefore removing the attribute from the system is not easily possible. A value between 0 and 1 implies that more APIs receive the attribute unnecessarily than consume it, while a value larger than 1 expresses the opposite situation. Especially the latter can be an indicator that the data attribute plays an essential role in the system. Splitting up its parent data structure – and the APIs returning it – could help reduce the data exposure. It should be noted that calculating this metric makes only sense for exposed data attributes. Hence, we assume that the denominator should never be zero.

6 Case Studies

To validate the explanatory power of our metrics, we evaluated our approach by conducting case studies of two open-source microservice reference implementations. Both projects were chosen because they provide a well-documented architecture and use many current technological standards and best practices. Despite not being real production systems, their maturity makes them a good alternative, also frequently used in other research studies [25–27].

6.1 Case Study 1: Lakeside Mutual

For our first study, we analyzed the excessive data exposure of the service communication within the 2020 Spring-Term edition of the *Lakeside Mutual*[3] project, a mainly Java Spring-based microservice system of a fictional insurance company. While the system consists of seven services, we focused only on the API-exposing functional services, resulting in a subset of five service implementations relevant to our research. Four used a Spring-based Java implementation, with the fifth one written in JavaScript. We also focused on concrete business use cases and ignored simple data reading and manipulation *CRUD* API operations [28] when possible, as they do not provide many insights from an architectural point of view. From a total of 31 analyzed APIs, our approach was able to identify five APIs with a positive $isEDE$ value, meaning these APIs expose at least one data attribute unnecessarily. Table 1 summarizes our findings:

The first API in the list, `CustomerCore/changeAddress` can be considered as an edge case, as it calls the DB for changing a customer's address but, in return, receives a complete customer entity as a result. While one could interpret this as an excessive exposure of data from a technical point of view – the API requests more data than it needs – we would not consider this call as a serious problem

[3] `https://github.com/Microservice-API-Patterns/LakesideMutual/tree/spring-term-2020`

Service	**API**	$EDE(v)$	$EDES(v)$
CustomerCore	/changeAddress	firstName, lastName, birthday, email, phoneNumber	moderate/high (but not relevant)
Policy Management Backend	/getPolicy	name	low
	/getPolicies	id, name	low
	/getInsuranceQuoteRequest	name	low
Risk Management Server	/handleClientRequest	firstname, lastname, streetAddress, city, email, phoneNumber, additional policy data	moderate/high

Table 1: Data attributes excessively exposed by *Lakeside Mutual* APIs

from a security perspective, since it does not happen between two microservice APIs but instead between a service and its underlying database, which we assume is a secure call within the service boundaries (see Section 3). Although reducing the exposure here could positively impact performance because of reduced message payload, we would not rate this exposure as security-critical.

The exposure detected in the following three APIs (from the *PolicyManagementBackend*) service originates mainly from data conversions between different API calls. In these specific cases, the composed `PolicyType` of a `PolicyAggregateRoot` is converted into a flat string representation for simplifying the data transfer. Our algorithm is not yet able to track all of these conversions, thus resulting in a false positive notification here.

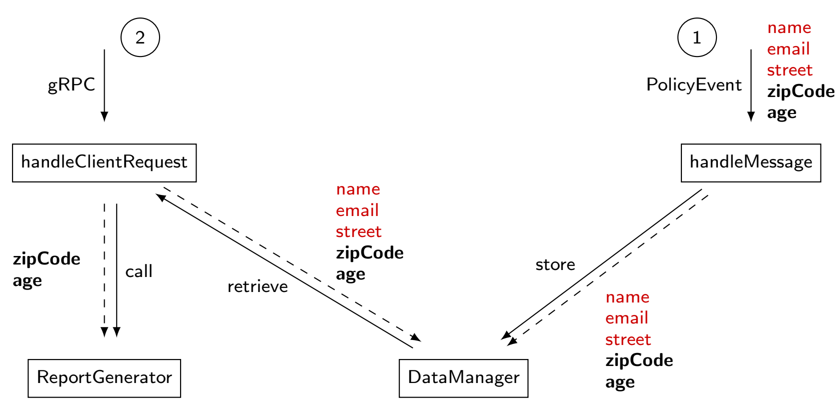

Fig. 4: Data exposure through API calls in the Lakeside Mutual Project. Not all of the incoming data attributes are eventually consumed, resulting in an excessive and unnecessarily exposure of customer information.

The last case is indeed more interesting as it is a two-staged process, illustrated by Figure 4: First, the *RiskManagementServer* reacts on incoming messages from the *PolicyManagementBackend* (1) and stores the received information in its internal data store. When an incoming gRPC (2) call (from a client application not part of our analysis) triggers the `handleClientRequest` API in the next step, the service starts with the generation of a report by accessing its internal data store. However, generating this report requires only a small subset of the stored data attributes, such as the customer's age or postal code. The remaining customer and policy data attributes the first API call received were stored redundantly and did not serve any purpose. While accessing the datastore from within the same process is again not problematic, here, the store itself acts only as a temporal buffer and the actual exposure already happens at the moment when the first API receives the data through an inter-service call.

Since most of the leaked data represent personal or contact information, we consider the $EDES(v)$ metric value of this exposure at least as *Moderate* or even *High* (see Section 5), especially regarding the large number of leaked attributes.

Two different strategies have been determined to avoid the exposure in this concrete case: (1) If other API endpoints do not consume these attributes, they can safely be removed from the API that initially exposed them. (2) If the attributes are instead also consumed elsewhere in the system, splitting up the initial endpoint into several smaller ones – each providing only the data for a specific use case – would be a better solution.

Calculating the $Usage - Exposure - Ratio$ (UER) for each exposed data revealed that despite a few exceptions, most of the exposed contact data (like the customer's phone number or street address) were not processed by any other API included in our analysis. However, skipping these potential redundant attributes from the whole system could be problematic as other client applications, which are not tracked by our approach, might still consume them.

So we suggest that splitting up these data-providing APIs into several, more use-case-specific ones would be the better strategy. In the concrete case, the `/getCustomer` API could therefore be separated into more fine grained endpoints, like, for instance, a `/getCustomerDataForReporting` and a `/getCustomerContactDetails` API.

6.2 Case Study 2: eShopOnContainers

As a second case study, we applied our approach to the *eShopOnContainers* system, Microsoft's reference implementation for a domain-driven-design-based microservice architecture[4]. The project imitates an online shop system consisting of several frontend components and a variety of different backend services, with the backend parts mostly communicating via an asynchronous event bus. For evaluating our method, we focused on the three backend microservices encapsulating the central domain logic and ignored other more infrastructure-related services like API gateways and authentication components. These three services provide 43 API operations, with the majority being asynchronous event handlers reacting on incoming event-bus messages.

[4] `https://github.com/dotnet-architecture/eShopOnContainers`, commit 59805331cd225fc876b9fc6eef3b0d82fda6bda1

Regarding the system's excessive data exposure, our study revealed the result outlined in Table 2: The first case in the table results from an event-handler that reacts to a `PriceChanged` event. Like the exposure we identified in our previous example, the API retrieves a complete entity from the underlying datastore but changes only a small subset of its attributes, making the requested remaining ones redundant. However, since this communication happens only between the service and its database, we do not consider this data exposure problematic. The second entry results from a false positive match. Here, incoming data is converted into a domain message through different conversion routines, and our detectors could not track the whole conversion process.

Service	**API**	$EDE(v)$	$EDES(v)$
Basket	ProductPriceChanged-IntegrationEventHandler	BuyerId, ProductName, Quantity, PictureUrl	low (not relevant)
Ordering	UserCheckoutAccepted-IntegrationEventHandler	ProductId, ProductName, Quantity, PictureUrl	low (not relevant)
Catalog	UpdateProductAsync	CatalogItem	low
	CreateProductAsync	PictureUri, AvailableStock, RestockThreshold, MaxStockThreshold, OnReorder	moderate

Table 2: Data attributes excessively exposed by *eShopOnContainers* APIs

We again had one false positive match regarding the *Catalog Service* due to direct database access from the API operation – instead of reaching out to the database through a repository. Since we had no detector for this kind of data access, our algorithm wrongly assumed that the receiving API did not process any incoming data attributes. The second match, however, indicates an actual case of excessive exposure: The `CreateProductAsync` API receives a new data object but uses only a subset of its attributes to create a new data entity. While these redundant attributes increase the payload of the data transfer, their impact on data security seems limited since none of the fields are processed further by the API. Still, we suggest replacing the current API parameter with a smaller variant containing only the attributes required for the creation process.

6.3 Summary

Our analysis revealed that both architectures are relatively secure regarding the excessive data exposure through their APIs. In most cases where more data was requested than eventually required, the transfer happened between the APIs and their underlying database. Since our research focuses on API-to-API communication, we assume that these connections are sufficiently secured and thus have not included them in our overall review. However, if such a secure database connection is not granted, the severity metric should be reevaluated to better reflect the security impact of these data exposures. We still identified two cases where data exposure between APIs happens and where this could indeed lead to unnecessary exposure of this data in case of a data breach.

7 Discussion

Based on our case studies, we can confirm **RQ1** and show for two non-trivial cases that it is possible to reconstruct a communication model of a microservice system and identify cases of excessive data exposure only by using static source code analysis. The upfront implementation effort for reading different language artifacts was also manageable. However, one limitation we see is that for providing support for dynamic or weakly typed languages we required some heuristics to correctly determine the data type of an API parameter. Our study also showed that our algorithm yielded some false-positive results due to insufficient detection of data conversion routines. Avoiding this would have required more specific detector implementation, which would have increased the implementation effort.

Regarding **RQ2**, we were able to show that our metrics are suitable for identifying data exposures and tracking the origin of these exposed data back to the initial data source. While our boolean or numerical metrics could easily be integrated into a dashboard, the more complex metrics would require a more sophisticated user interface. We think of a graphical representation where expert users could mark a specific data attribute and follow the communication call graph to its origin. As we are not aware at the moment of any visual system like that, it would certainly represent an interesting option for further research.

Considering **RQ3**, our case studies demonstrated how our metrics can guide architects in redesigning a microservice system towards better data security. Based on the findings our metrics provided, we can identify accidentally exposed data and reorganize the API structure to minimize these exposures, thus reducing the damage caused by potential data leaks. However, data security is only one aspect that affects API granularity, and other factors, like maintainability or performance, must also be considered, too [19]. Also, microservice architectures are often based on or interact with older legacy systems, which can constrain their API structure. Balancing out all these aspects is not a trivial task and requires sophisticated domain knowledge. Therefore, although possible, we would advise against a fully automated refactoring process and see our approach more as a supporting tool during the decision process.

8 Threats to Validity

To ensure our case study accounts for potential bias or unintended factors, we tested our approach against commonly used threats that might influence the validity of our findings as suggested in [29].

Construct Validity While most of our detectors operate directly on the source code, we also applied heuristics at some places to simplify their implementation.

To ensure that our model adequately represents the actual system, we added several manual verification steps. Nevertheless, the case studies showed that it is still possible to miss some code artifacts, but we were able to detect and fix these cases manually.

Internal Validity Although our research focuses on data security, we know that various other factors strongly impact API (re)design. Thus, we present our approach as one possible tool to guide the decision-making process, but other aspects must also be considered for the final decision.

External Validity Since the data transfer heavily depends on the concrete business domain or use case, finding a reference system across several domains is extremely difficult. Still, we think the projects we chose for our study provide a good compromise. They are actively developed and utilize many best-practices implementations used among other systems. Several authors have also identified data leaks and breaches as a significant problem in microservice architectures (see, for instance, [2, 6] or [14]). Therefore, our research provides a significant contribution to reducing the damage caused by these leaks.

9 Conclusions and Future Work

Our paper presents a novel approach for tracking excessive data exposure between microservice APIs. Data is excessively exposed whenever an API receives more data than it actually consumes, causing an unnecessary and avoidable risk for potential data leaks. To make this amount of exposure measurable, we introduced a technique to reconstruct a communication and data flow model from underlying source code artifacts. Based on this model, we defined a set of architectural metrics to identify and evaluate API calls that unnecessarily expose data attributes. We verified our solution on two case studies and, based on our findings, suggested guidance on how the APIs could be restructured to minimize the amount of data being unnecessarily exposed.

While our approach can identify many of the most common matters where data is exposed, some edge cases were not within our scope, e.g., the amount of data leaked through service error handling [5]. Despite data security, other aspects exist that influence the granularity of APIs. Incorporating all these aspects into a more holistic decision model for API granularities would undoubtedly be a promising field for further research.

Acknowledgments: This work was supported by: FWF (Austrian Science Fund) projects API-ACE: I 4268 and IAC^2: I 4731-N. Our work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647 (AssureMOSS project).

Bibliography

[1] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[2] L. Miller, P. Mérindol, A. Gallais, and C. Pelsser, "Towards secure and leak-free workflows using microservice isolation," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2021, pp. 1–5.

[3] D. Yu, Y. Jin, Y. Zhang, and X. Zheng, "A survey on security issues in services communication of Microservices-enabled fog applications," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 22, Nov. 2019.

[4] A. Hannousse and S. Yahiouche, "Securing microservices and microservice architectures: A systematic mapping study," *Computer Science Review*, vol. 41, p. 100415, 2021.

[5] W. K. A. N. Dias and P. Siriwardena, *Microservices Security in Action*. Simon and Schuster, 2020.

[6] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," *arXiv preprint arXiv:2103.00170*, 2021.

[7] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, 2016.

[8] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[9] T. Yarygina and A. H. Bagge, "Overcoming Security Challenges in Microservice Architectures," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. Bamberg: IEEE, Mar. 2018, pp. 11–20.

[10] E. Ntentos, U. Zdun, K. Plakidas, D. Schall, F. Li, and S. Meixner, "Supporting architectural decision making on data management in microservice architectures," in *European Conference on Software Architecture*. Springer, 2019, pp. 20–36.

[11] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Microservices in industry: insights into technologies, characteristics, and software quality," in *2019 IEEE international conference on software architecture companion (ICSA-C)*. IEEE, 2019, pp. 187–195.

[12] A. Singjai, U. Zdun, O. Zimmermann, and C. Pautasso, "Patterns on deriving apis and their endpoints from domain models," in *26th European Conference on Pattern Languages of Programs*, 2021, pp. 1–15.

[13] P. Genfer and U. Zdun, "Identifying domain-based cyclic dependencies in microservice apis using source code detectors," in *European Conference on Software Architecture*. Springer, 2021, pp. 207–222.

[14] X. Shu, D. Yao, and E. Bertino, "Privacy-preserving detection of sensitive data exposure," *IEEE transactions on information forensics and security*, vol. 10, no. 5, pp. 1092–1103, 2015.

[15] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87–97, 2009.

[16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[17] J. Soldani, G. Muntoni, D. Neri, and A. Brogi, "The μtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures," *Software: Practice and Experience*, 2021.

[18] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 254–265.

[19] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service-and microservice-based systems: a literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, 2017, pp. 107–115.

[20] I. Saidani, A. Ouni, M. W. Mkaouer, and A. Saied, "Towards automated microservices extraction using muti-objective evolutionary search," in *International Conference on Service-Oriented Computing.* Springer, 2019, pp. 58–63.

[21] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, and M. Stocker, "Interface responsibility patterns: processing resources and operation responsibilities," in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, 2020, pp. 1–24.

[22] A. Walker, D. Das, and T. Cerny, "Automated code-smell detection in microservices through static analysis: a case study," *Applied Sciences*, vol. 10, no. 21, p. 7800, 2020.

[23] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, and W. Hasselbring, "Detector-based component model abstraction for microservice-based systems," *Computing*, vol. 103, no. 11, pp. 2521–2551, 2021.

[24] L. Fan, Y. Wang, X. Cheng, and S. Jin, "Quantitative analysis for privacy leak software with privacy petri net," in *Proceedings of the ACM SIGKDD Workshop on Intelligence and Security Informatics*, 2012, pp. 1–9.

[25] F. Rademacher, S. Sachweh, and A. Zündorf, "A modeling method for systematic architecture reconstruction of microservice-based software systems," in *Enterprise, Business-Process and Information Systems Modeling.* Springer, 2020, pp. 311–326.

[26] H. Vural and M. Koyuncu, "Does domain-driven design lead to finding the optimal modularity of a microservice?" *IEEE Access*, vol. 9, pp. 32 721–32 733, 2021.

[27] A. El Malki and U. Zdun, "Evaluation of api request bundling and its impact on performance of microservice architectures," in *2021 IEEE International Conference on Services Computing (SCC).* IEEE, 2021, pp. 419–424.

[28] A. Mashkoor and J. M. Fernandes, "Deriving software architectures for crud applications: The fpl tower interface case study," in *International Conference on Software Engineering Advances (ICSEA 2007).* IEEE, 2007, pp. 25–25.

[29] R. K. Yin, *Case study research and applications.* Sage, 2018.