Extending traditional request logging

what developers require from logging solutions and anonymization of privacy sensitive information in logs

Oostdam, S.N.W.

Award date:
2019

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science

Extending traditional request logging: what developers require from logging solutions and anonymization of privacy sensitive information in logs

Master Thesis

Stijn Oostdam

Supervisors:
dr. D. (Dirk) Fahland
dr. N. (Nicola) Zannone
F. (Frits) van de Water

Publication Version 13-05-2019

Eindhoven, May 2019

Abstract

This research aims to extend traditional request logging in order to aid developers when solving complex bugs while protecting privacy sensitive information in those logs. A software bug is an error in a computer program that causes incorrect results or behaviour. Software developers that need to fix bugs often refer to logs to get insights on program and user behaviour around the time a bug occurred. These logs can show the cause of the bug or have data in them that helps developers reproduce the bug. However, complex or data related bugs cannot always be solved with the help of logs. In those cases, more data is needed to find the cause of and fix the bug.

Het NIC is a financial services company that is currently developing a web application for their complex business process. This web application sometimes has bugs which are only reproducible on production environments. The current basic request logging, where every request to the server is logged, is not sufficient to solve such complex bugs. This research aims to find a way to extend the logging of the web application to make complex bugs easier to solve.

However, this web application contains privacy sensitive invoices of other companies. The contracts *Het NIC* has with customers whose invoices are in this web application define that only a specific set of people are allowed to access these invoices. These contracts demand that no production data is seen by anyone outside the set of approved persons. Therefore, the logs and all related data must be anonymized before developers can access it.

The first step in extending logging was to discover what developers require from logging solutions and how they utilize information in logs. We interviewed developers and found out that they would like to see as much information as possible in the context of a server request. Furthermore, they would like to have the ability to retrace the steps users have taken just before and at the time a bug occurred. Having this information helps them in reproducing and discovering the cause of a bug. We formulated the requirements for a new logging system from their answers, combined with the technical requirements stated by *Het NIC*. The main requirement was to capture and log all data that is communicated between user and server.

Similarly, the second step was organizing a group discussion around all different data types (booleans, string, numbers, etc.) in logs in order to discover how developers would like to see each data type anonymized. Their wishes were analyzed and the realistic ones were transformed into the requirements for the anonymization.

When all requirements were defined, we started the realization of the new logging system. First, we chose ElasticSearch [5] for our log storage combined with Kibana [5] front end. Around ElasticSearch, we designed the logging system, which includes a .NET Core proxy that is responsible for the anonymization and a library that logs all data that can be included in any web application.

The next step was to implement the anonymization algorithm. We chose to apply permutation to all data by default in order to guarantee the highest possible level of anonymization. Due to the requirements, this permutation has some exceptions and therefore not everything is anonymized. For example, we preserve string and number length and NULL and boolean values. We also introduced dynamic anonymization, in which the user can query whether a specific field in log data was between certain ranges. The risks that come with the preservation and dynamic anonymization were analyzed and were solved or deemed insignificant.

Lastly, we performed an empirical evaluation to test whether the developers' ability to fix bugs has improved with the new logging solution. We recreated bugs from the past and gave developers the task to fix them with the help of the anonymized log data. Not all bugs were solvable due to the information loss in anonymization, which means that there will always be bugs which need exact information to be solvable. However, the evaluation showed that the extra information helps a developer localize, reproduce and solve a bug. Therefore, the extended logging does help developers in understanding bugs but not with all different kind of bugs.

Acknowledgements

Firstly, I want to thank my committee members: dr. Dirk Fahland, dr. Nicola Zannone and Frits van de Water. I want to thank Dirk Fahland for supervising me during the complete graduation project. Your input, guidance and feedback were really helpful and brought this graduation project to a successful result. I also want to thank Nicola Zannone for giving input around ideas with me and Dirk Fahland.

Next to this, I want to thank *Het NIC* and especially Frits van de Water for providing me with the opportunity of graduating at *Het NIC* around the interesting subject of logging. Furthermore, the developers at *Het NIC* deserve a thank you for taking the time to be interviewed, hold a group discussion and test the new logging solution. I also want to thank the other developers who took the time to be interviewed.

Finally, I want to thank my family and friends for their support throughout this graduation.

Stijn Oostdam
Eindhoven, May 2019

Contents

1 Introduction

This master's thesis researches logging data access in web applications in order to help developers in their bug fixing process. It focuses on determining which information developers need from logs to solve bugs and how privacy sensitive data in logs can be protected. This section will give an overview of the problem.

1.1 Problem context

Het NIC is a financial services company that checks invoices as their main business and facilitates this graduation project. The company inspect ledgers of other companies for errors such as double payments. *Het NIC* also has daughter companies which focus on other businesses such as selling office supplies and facilitating home care to patients. These companies all have custom built applications in use.

Currently, the process of checking invoices is performed using Microsoft Excel and Microsoft Access. However, these applications are reaching their limits on functionality and storage levels. Therefore, *Het NIC* has a team of software developers who are developing a new web application which will replace the current systems over time. This web application is called Accounts Payable Recovery Audit (APRA) which is developed in ASP.NET and uses the Model-View-Controller design principle. A pilot version of APRA is already being tested by the users.

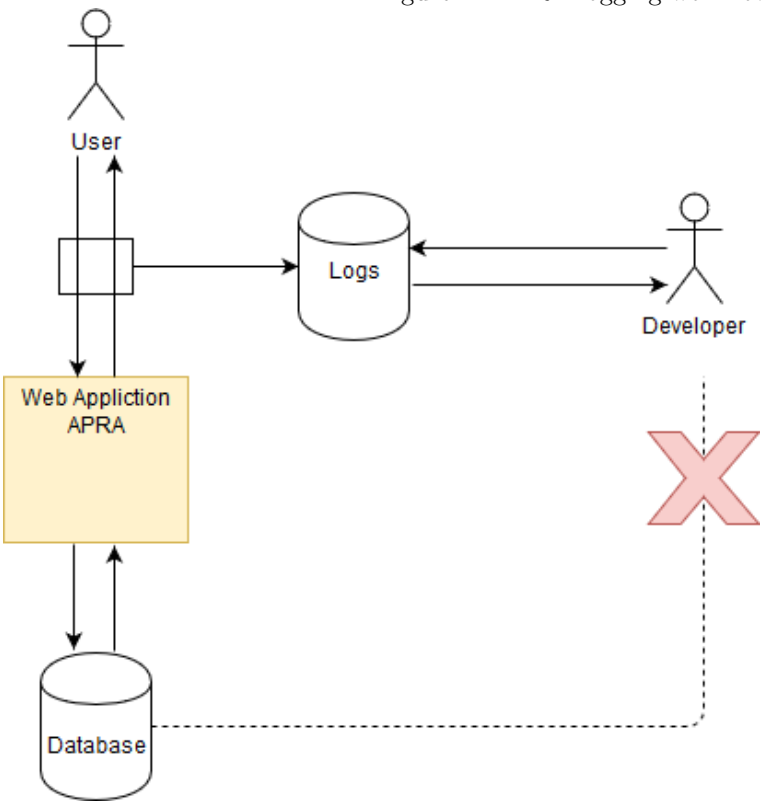
In the past, *Het NIC* encountered bugs in their software, which could not be reproduced by developers. These bugs were caused by specific data in live environments, which was not available in develop or test environments. Giving developers access to their live environments is something *Het NIC* wants to prevent, since these environments always contain privacy sensitive data. Granting access to a single developer can be a process of multiple months where the developer has to pass different kinds of background checks. Therefore, *Het NIC* is searching for a way to record more information about the live environment so a developer does not need to access the live environment to solve bugs.

The pilot version of APRA has the following logging in place. For each request, the time, URL and method are being logged. Exceptions and their stack traces are logged as well. Furthermore, developers can implement custom log statements on crucial parts of their code. Both the automatic and manual log statements are written to a text file on the server. Application Insights [3] is also implemented to monitor the system health and performance during the pilot.

The basic logging in the pilot version generally gives enough information for developers to solve bugs. The logging gives them an idea of the actions a user has taken around the time a bug occurred. However, in the case of a bug where specific data is involved, the logs are insufficient. The current situation is shown in Figure 1. As can be seen, the developer does not have access to the production database and never should get access. Thus, the main objective of this thesis is to answer the following question:

How to extend basic logging such that developers are provided with more information about a live environment when solving bugs, while preserving privacy of data in logs?

Figure 1: APRA logging workflow



1.2 Problem description and approach

This section will discuss the problems this thesis addresses, which will be split into three research questions. The approach to answer each research question will be explained.

1.2.1 Research question 1: what developers require from logs

Logging can be extended in many different directions. However, not every piece of information will be relevant for developers when solving bugs. Therefore, in order to extend the logging functionality to help developers when solving bugs, it must first become clear what data and in which form developers actually need from logs to solve bugs. Thus, the first research question is:

What kind of data and in which form do developers need when solving bugs?

Section 2.1 researches the most common use cases of logging and, as an extension, Section 3.1 discusses interviews that are conducted with a variety of developers. These developers give insights in their experience with data-related bugs as well as their ideal logging situation. It becomes clear that they preferably have access to as much data as possible related to that bug. Therefore, the basic intuition is to extend the request-logging with the data the user has seen on their screen.

That way, data from the live environment is captured and visible for the developers in the context of a request.

1.2.2 Research question 2: designing a logging system

The current logging of APRA requires undesirable Remote Desktop (RDP) access to the server in order to open the log text file. Furthermore, while APRA is the main focus of this thesis, *Het NIC* would like to have a universal logging solution for all their web applications. Such a solution should have ordinary request and exception logging extended with logs of what the user has seen, in order to aid developers when solving difficult bugs. Realizing these requirements will require setting up and designing a complete new logging system and includes a number of technical challenges such as modularity, performance, log volume and efficient log querying. As stated, the privacy sensitive information that can appear in the logs should be protected. The second research question is as follows:

How to design a logging system that can capture what a user has seen and protect privacy sensitive information in logs?

Section 2.1 researches what the most common problems and challenges are with logging. Thereafter, Section 3.1 combines the developers' requirements, the logging challenges and the technical demands from *Het NIC* into requirements for a new logging solution. Section 4 then presents a complete new logging system that can be deployed for any web application of *Het NIC*. It is able protect privacy sensitive information in logs by applying the chosen anonymization technique when developers want to view logs.

1.2.3 Research question 3: privacy sensitive information in logs

The web applications of *Het NIC* all work with sensitive information. In the domain of APRA, all data is based on invoices. These invoices have a fixed structure, but can be represented in many ways through the post and return data. In their full representation, an invoice contains privacy sensitive fields like company/person names, addresses, personal identification numbers, IBANs, etc. This means these invoices contain information about companies but sometimes information on natural persons as well. All other data in the system is generated in support of these invoices. For example, there is an entity called "grouped creditor" which is an entity that represents multiple creditors from the invoices and therefore these entities also contain privacy sensitive information. Therefore, at all times, logging user activity and the data a user has seen implies that privacy sensitive data is being logged. An example of invoices with sensitive information is shown in Listing 1 in JSON format.

Listing 1: List of basic non-anonymized invoices

```
{
  "Id": 4028,
  "EntityNr": "10",
  "CreditorNr": "358",
  "System": "Exact",
  "CreditorName": "SomeCompany B.V.",
```

```
  "InvoiceDescription": "Advice and guidance sales Customer1 and Customer2 GMGH",
  "Address": "Woudenbergselaan 101",
  "Postalcode": "3943 ME",
  "City": "VOLENDAM",
  "IsRisk": true
},
{
  "Id": 4063,
  "EntityNr": "10",
  "CreditorNr": "358",
  "System": "Exact",
  "CreditorName": "SomeCompany B.V.",
  "InvoiceDescription": "Credit advice and guidance sales Customer1 and Cus2",
  "Address": "Woudenbergselaan 101",
  "Postalcode": "3943 ME",
  "City": null,
  "IsRisk": true
},
{
  "Id": 4118,
  "EntityNr": "10",
  "CreditorNr": "360",
  "System": "Exact",
  "CreditorName": "Some Holding BV",
  "InvoiceDescription": "Advice and guidance sales Cus1 and Customer 2 GmbH",
  "Address": "Woudenbergselaan 101",
  "Postalcode": "3943 ME",
  "City": "VOLENDAM",
  "IsRisk": false
},
{
  "Id": 4127,
  "EntityNr": "10",
  "CreditorNr": "360",
  "System": "Exact",
  "CreditorName": "Some Holding BV",
  "InvoiceDescription": "Credit 170504",
  "Address": "Woudenbergselaan 101",
  "Postalcode": "3943 ME",
  "City": "VOLENDAM",
  "IsRisk": false
}
```

Logging information such as these invoices will be subject to international and national laws. Firstly, recording privacy sensitive information is subject to the General Data Protection Regulation (GDPR) [6]. Furthermore, APRA will be used by employees of Het NIC, which means that approval of the works council (Ondernemingsraad) is needed according to the Dutch law [13], since the logging could be used to monitor the employees. That law also holds when an organization does not intend to use it for employee monitoring. Lastly, the Autoriteit Persoonsgegevens (AP) [4], the Dutch Data Protection Authority, has announced that, under the GDPR, it can and will perform periodic checks to see if companies do not violate data access and usage rights. Recording user activities and data they saw could help prove *Het NIC* to the AP that it operates within the law.

Considering the GDPR, *Het NIC* has contracts with customers which are more strict than the GDPR, meaning the contracts can include more strict policies about handling the customers' data. In each contract with a customer, only a certain set of people are given the clearance to work with the customers data and developers are never included in that set of people. Therefore, the data in the logs need to be anonymized when developers request to view them, as they are not allowed to see the real data according to the contract. The challenge in anonymizing these logs is to guarantee full removal of sensitive information while preserving its usefulness for debugging. Anonymization of the invoices from Listing 1 means that a creditor (field CreditorName) is no longer directly recognizable and no external data can be used to perform any attacks to discover which creditor or person appears in the data. As can be seen, other fields like InvoiceDescription can also contain names which should not be recognizable after anonymization.

Thus, the third research question, with three sub questions, is:

How to give developers access to privacy sensitive logs such that privacy is preserved?

- **What kind of privacy sensitive data occurs in the logs?**
- **How to apply an anonymization technique such that the utility of the logs is preserved?**
- **How to apply an anonymization technique that works with any data structure?**

The goal of these questions is to discover a way to shield the privacy sensitive information in logs from developers while the logs should stay useful for the purpose of solving bugs.

However, firstly, the goal of these questions is not to meet the privacy requirements of every contract *Het NIC* has with its customers. These contracts differ each time and it is unfeasible to cover each and every aspect. For example, a contract could need extra organizational measures next to the anonymization, such as explicit permission of a project manager before viewing any log data. Instead, the goal is to provide a solution which is sufficiently strict for most contracts and where organizational matters can be enforced when needed.

Secondly, the goal is not to be GDPR compliant, i.e. ensuring that the solution and *Het NIC* operate fully within that law. Making a solution GDPR compliant is not a simple case of taking technological security measures. Naturally, the provided solution will have standard security measures in place, but GDPR compliance is not guaranteed. Ensuring GDPR compliance involves appropriate business processes next to the technology being used.

Lastly, the goal is not to comply to the Dutch law for work councils and neither to meet the requirements the Autoriteit Persoonsgegevens has. Complying to every contract, to the GDPR, to

the work councils law and to the AP concerns privacy management around the logs themselves and is not part of this thesis.

To answer the third research question, firstly, Section 2.2 researches the possible anonymization techniques, attack methods on anonymized data and methods to measure the degree of anonymization. Research into these topics is needed to design a robust anonymization method.

Secondly, the kinds of privacy sensitive data in the logs depends on which data types can be recorded. Section 3.2 discusses the results of group discussion held with developers to discover whether they think a data type is privacy sensitive and which information of that data type they deem important.

Finally, Section 5 then presents an anonymization solution, which is able to apply anonymization on different data types regardless of the data structure.

1.3 Results

The problem *Het NIC* tries to solve requires that in some way the existing logging is extended in order to help developers when solving bugs when they cannot be reproduced locally. The first intuition is to log what a user has seen on their screen. For example, if the invoices of Listing 1 are displayed to a user, it means that they will be logged, including their privacy sensitive information.

The contracts *Het NIC* has with customers often require that only a certain set of people have access to their data, and developers are not part of that set. This means that the data in the logs should be anonymized. The realization of an anonymization technique should be able to anonymize the data in the example of Listing 1, but also data where the structure is not known beforehand.

Section 4 discusses the implementation of a new logging system. First, we chose ElasticSearch [5] for our log storage combined with Kibana [5] front end. Around ElasticSearch, we designed the logging system, which includes a .NET Core proxy that is responsible for the anonymization and a library that logs all data that can be included in any web application.

Section 5 discusses the anonymization technique and we chose to apply permutation to all data by default in order to guarantee the highest possible level of anonymization. Due to the requirements, this permutation has some exceptions and therefore not everything is anonymized. For example, we preserve string and number length and NULL and boolean values. We also introduced dynamic anonymization, in which the user can query whether a specific field in log data was between certain ranges. The risks that come with the preservation and dynamic anonymization were analyzed and were solved or deemed insignificant.

Lastly, we performed an empirical evaluation to test whether the developers' ability to fix bugs has improved with the new logging solution in Section 6. We recreated bugs from the past and gave developers the task to fix them with the help of the anonymized log data. Not all bugs were solvable due to the information loss in anonymization, which means that there will always be bugs which need exact information to be solvable. However, the evaluation showed that the extra information helps a developer localize, reproduce and solve a bug. Therefore, the extended logging does help developers in understanding bugs but not with all different kind of bugs.

2 Background

This section gives an overview of available literature, techniques and challenges concerning logging and anonymization. Having an overview will help making decisions regarding the usage of technologies and techniques for logging and approaches for anonymization.

2.1 Logging

Logging is the programming practice of storing important program runtime information. Developers can add log statements to the source code themselves or can decide to use an existing logging system. Such a log statement often consists of a verbosity level (e.g. trace/error/fatal), a message and variables. Logs are generally stored to record system behavior for analysis of problems related to function, behavior or performance of the system.

In this graduation project, a new logging system will be set up for *Het NIC*. Therefore, Section 2.1 first discusses the different use cases of logging in order to get an understanding of what developers need and expect from a logging system. We further research methods and challenges to know where to pay attention to when designing the system. The challenge of log storage is discussed extensively since the choice for a storage technique will be the first to make and it affects all further development.

Use cases

Here, the most common and well-known use cases are outlined to discover the basic requirements of a logging system.

Error debugging: perhaps the most important use case of logging is error debugging. Conventional debugging and stepping through lines of code is often not possible in (large-scale) live applications [25] and developers have to combine the error report with available log information to solve the problem.

System monitoring: another use case for logging is the monitoring of system performance and health. In the case of web applications, such logs would include the time it takes to complete requests, CPU usage per server, RAM usage per server, amount of exceptions in a certain time period, etc. These statistics enable live tracking of system performance, with which system administrators can make decisions such as scaling the system resources up or down. Historical logs can be analyzed to identify bottlenecks in the system [32, 40] or to check the system health at the time an error occurred.

User activity logging: user activity logging is a technique to get insight in how users use an application. In web applications, user activity logging starts at recording each request send by the user. It can go as far as capturing all interactions made by users [14]. This information is useful to analyze web applications for ways to improve user experience.

Trace multi-service code execution: researchers have also utilized logging to trace the execution of code in multi-service applications and networks. A system introduced in [17] adds log statements to different levels in the network (application, session, transport layer). [15] explains a method to track database transactions through the layers of a web application, the web servers, application server and database servers. Similarly, Google developed tracing for large-scale distributed systems [41].

Anomaly detection: a useful application of log analysis is anomaly detection. Anomalies can be performance problems, work flow errors, etc. For example, [27, 19] both utilize log mining to

determine common program work flows and can detect process anomalies with those work flows.

Usability analysis: as mentioned before, it is possible to measure the usability of interactive applications using logs. Query logs can be used to discover which tasks in a program users find difficult to perform [18] and those tasks could then be simplified and optimized.

Other examples of logging use cases include retrieval evaluation [24], search recommendations [38] and predicting cost-performance trade-offs [35].

Since we will design a new logging system for the purpose of debugging and monitoring user activity, it should support functionality for error debugging and user activity logging. Monitoring of system performance and health will not be included since *Het NIC* already uses Application Insights. Furthermore, APRA is an application consisting of one service so therefore multi-service tracing is not needed. In the future, usability measures could be added by querying the user activity logs.

Methods

In general, there are two methods to log, automatic and manual. Automatic logs are written by software components included in the system, such as a database or web server. Manual log statements are written by developers. Developers often include log statements when validating variables, handling exceptions and inside logic branches [20], but are free to write a log statement anywhere in the source code at places they deem crucial or prone to error. It is preferable that writing a log statement costs low effort for developers such that they are inclined to log more. A typical easy-to-use log statement is of the following form: `Logger.Write(Level, Message, Variables)`. Furthermore, it is a best practise to have as much automated logging as possible, since developers use free text in their log statements, which can make them harder to understand and parse.

Therefore, in this thesis, we will explore automated logging. User activities and the data on views should be automatically logged and not require any programming from developers.

Challenges for writing and reading log entries

There are numerous challenges regarding logging, including usage, storage and security [22, 31, 34]. These challenges need to be identified such that they can be taken into account when designing a new logging system.

Developers writing log statements: log statements written by developers have several risks. Firstly, when logs are used for debugging, there is a risk that developers use too general or unclear language or an incorrect severity level, which increases the difficulty to locate the right set of log events related to a problem. Secondly, log statements of different threads can be intertwined. Thirdly, a log could miss the variables related to a problem. Lastly, logging too little information results in useless logs, while logging too much information can make it more difficult to query logs and extract information and increases the log volume. Though automated log events have less of these problems, it will always remain difficult to select which variables to log due to the diverse nature of logs.

Log volume: there is the challenge of finding the perfect balance between usefulness of log events and their volume. For a developer, it is attractive to log as much information as possible. However, as mentioned before, logging more is not guaranteed to increase the usefulness. Logging

more information also requires more storage and has more overhead during program execution. Developers need to have good reasons to include frequent and/or expensive log statements.

Privacy sensitive: recording user activities in logs is deemed as privacy sensitive information and its storage and usage therefore is subject to national and international privacy laws. While having this data can make an organization competitive, the storage server and the data need to be strongly secured, since this kind of data could be misused by employees or other people with malicious intents. Such international law is the GDPR [6], which enforces an organization to have valid reasons to collect this data as well as provide sufficient storage security and data retention plans. For example, collecting user information to improve the user experience can be a valid reason, but using that data for any other purpose is then strictly forbidden. Articles in the GDPR related to this thesis are data collection, data security, data retention, account and data deletion (customers of *Het NIC* are the data owners) and privacy breach notification [45].

Log storage: traditionally, log events have been stored in text files. Each line in the file represents one log event. While those files continue to work for smaller applications, they have numerous drawbacks when used for larger systems. Text files do not scale well with higher log volumes, it is difficult for developers to search through those files and they are far from ideal if the systems consist of multiple servers. Through the years, much research has been done to improve log storage.

Similar to the use cases of logging, there is a large variety of research done in the storage. Secure logging is one of the topics that has gained attention, primarily due to the privacy-sensitive nature of log events. For example, [28] talks about a scheme to verify all log entries and [16] introduces a system for privacy preserving search logging. Recent research goes as far as storing log files on a blockchain to guarantee that they are not tampered with [36].

Other research includes focus on a higher-level logging architectures [30] and large-scale logging [37].

A market leader in log storage is Apache Lucene [2], which presents itself as high-performance, full-featured text search engine library. Lucene is especially useful for high-volume logging, as it claims to be able to index over 150GB of text per hour and have powerful and efficient search algorithms. Therefore, Lucene's main purpose is log storage, but it can be used for any other task that requires large text indexing and searching. Lucene is released under the Apache Software Licence, making it free for commercial use.

However, using Lucene within an organization is difficult since it has no front end to issue and view queries. That is why solutions like ElasticSearch [5], Algolia [1], Splunk [11], etc. exist. These solutions typically release free versions with basic functionality and have (costly) commercial licences with full feature packs. ElasticSearch' popularity made the definition ELK-stack well known. ELK stands for ElasticSearch, Kibana and Logstash, which are all products related to log management. It means a stack of log ingestion (Logstash), log storage (ElasticSearch) and log querying (Kibana). Commercial log management solutions aim to offer a complete ELK-stack as a product. More recently, free and commercial log storage solutions have gained popularity, since it is costly for organizations to implement their own custom log storage technology. Looking at the feature list of ElasticSearch (querying, analysis, speed, scalability and resiliency), implementing and maintaining such a system as an organization requires much time and much knowledge across multiple fields of expertise.

In this thesis, we chose to use an existing storage technology due to complexity (and therefore time) of implementing storage technologies. Furthermore, the required knowledge for building a storage system is simply not available. All other challenges will be taken into account when

designing the logging system.

2.2 Anonymization

In the context of this thesis, anonymization is a type of information sanitization with the intent to protect privacy. Anonymization is the process of mutating or removing personally identifiable and privacy sensitive information. This way, the persons and companies whom are described by the data remain anonymous when releasing the data. Next to anonymization, there is pseudonymization. Pseudonymization is the process of replacing personally identifiable information by one or more artificial identifiers, pseudonyms. The big difference between anonymization and pseudonymization is that pseudonymized data can be restored to its original state with additional information by using the artificial identifiers, while anonymized data can never be restored.

Personally identifiable information is any information that can uniquely identify a natural person. This information can be a name, identification number, IBAN, etc. Next to that are quasi-identifiers [43], which are attributes that together as a set can identify a person. A good example of quasi-identifiers is that 87% United States citizens are uniquely identifiable by their ZIP, gender and date of birth [42]. Releasing data anonymized means that all identifiers and quasi-identifiers are altered in some way such that, even with the inclusion of outside data sets, the natural persons are no longer (uniquely) identifiable. The above statement also holds when dealing with privacy sensitive information from companies.

This section explores existing anonymization approaches, attacks to anonymized data and techniques to measure anonymization. This exploration will help in discovering how to anonymize data in logs, identify possible attack vectors to that anonymized data and find ways to measure the quality of the anonymized data.

Research that specifically addresses anonymization for the purpose of debugging and solving bugs is sparse and therefore not included in this section. This problem will be addressed in Section 3.2, where research will be done to discover what information in logs developers find important in order to make better decisions on how to anonymize logs.

Anonymization approaches

Which anonymization approach to use depends on the type and use of data. There is tabular data, item sets and graphical data [23]. Approaches focusing on tabular data are of interests, since *Het NIC* solely uses data in that form. Research proposed five approaches to the anonymization of tabular data [21]:

- **Generalization**: privacy sensitive data is generalized such that the person is not identifiable but the data is still useful. For example, a specific age is generalized into an age range or a zip code is generalized such that it no longer present a specific address but a city. A well known anonymization technique that leverages generalization is k-anonymity [43].
- **Suppression**: an identifier is completely removed from a dataset. In the case of a set of quasi-identifiers, quasi-identifiers need to be removed until that set no longer uniquely identifies a natural person. The process of removing is either removing attributes from the dataset or replacing values of attributes with other (random) values.
- **Perturbation**: anonymization by perturbation is the addition of noise to attributes. This addition should always be random, otherwise noise patterns could be discovered which com-

promises the data. Perturbation is often used for statistical studies on data, where the addition of noise does not impact the statistical value from the data.

- **Permutation**: Altering the sensitive data such that the privacy sensitive information is no longer recognizable. Since data is altered, permutation can cause information loss.
- **Anatomization**: anatomization disassociates relationships between (quasi-)identifiers and sensitive attributes. More precisely, a (quasi-)identifier-table and a sensitive-data-table are released separately. Records in both tables belong to groups instead of unique records, such that sensitive information can no longer be linked to a unique identifier.

Which method or combination of methods is most suitable to use when anonymizing logs will be discussed in Section 5.

Attack and Privacy models

When an organization releases privacy sensitive data, this data can be vulnerable to one or more attack models. There are numerous privacy models which are designed to protect against certain attack models, however there is no privacy model which covers all attack models. Therefore, the organization must carefully consider in which way their data is vulnerable and choose the correct privacy model.

The record linkage attack model [21] relies on the discovery of small groups records which share the same quasi-identifiers values. For example, this discovery can happen when quasi-identifiers are not generalized enough. Having a quasi-identifier value which links to a small group of makes that group vulnerable to attacks with the addition of outside information. One way to protect against record linkage is through the notion of k-anonymity [43]: if one record in the released data has some value qid for its quasi-identifiers, at least $k-1$ other records have the same qid value, which results in a probability of $1/k$ of linking a specific record through its quasi-identifiers. k-anonymity is not perfect and many research has been done to improve it, like (X, Y)-anonymity [46].

The attribute linkage attack model [21] does not give an attacker the exact record of a target, but could find sensitive information with relatively high confidence. Again, consider a small group of records which share the same quasi-identifiers values, then sensitive information can be compromised if the distribution of sensitive information within that group is not diverse enough. The researchers of [29] introduced a notion called l-diversity, which requires each group of records which share the same quasi-identifiers values to contain at least l distinct sensitive values. However, while released data can satisfy a certain l-diversity value, if the overall distribution of the sensitive values is skewed, it does not help in preventing an attribute linkage attack. Researchers tried resolving this problem with t-closeness [26]. These techniques are not the only ones which address the problem of attribute linkage and it is still an important research topic to this day.

The table linkage attack model [21] assumes that it is enough to know if a specific person is present or absent in the released data. To execute a successful record or attribute linkage attack on a specific individual, an attacker must know that the individuals data is in the released data. However, this is not a requirement for a table linkage attack. Using an external public table, it is possible to calculate the probability a person is in a released data set. δ-presence [33] aims to give a measure to this probability and discusses how to lower it. The downside of δ-presence is that it assumes the data publisher has access to the same external public table as an attacker, which does not always hold in practise.

The last group of privacy models focuses on how the probabilistic belief on the sensitive information of an individual is changed when anonymized data is published [21]. For example, one study defines a notion to measure the increase in risk to a data owner's privacy when their information is published in a statistical database. Other researchers suggest that having access to published anonymized data should not enhance an attackers power of isolating any record owner.

In the most basic form, *Het NIC* works with invoices which contain information about companies. Releasing this information anonymized can be subject to table, record or attribute linkage attacks if an attacker has access to a similar dataset as *Het NIC*. Therefore, all described attack models should be taken into account when designing the anonymization method in this thesis.

Measuring anonymous data

The privacy models described above all introduced a measure of anonymity of published data. For example, published data is k-anonymous when its probability to link it to a specific record is $1/k$. However, measuring the utility of data after being anonymized is important as well, i.e. measuring the quality of anonymized data. Often, the data publisher does not know for which purpose, e.g. statistical analysis, the anonymized data will be used. If the data publisher does know the purpose, specialized measurements can be designed.

In the case of an unknown purpose, a useful metric can be to measure the similarity between the original and released data, a principle of minimal distortion [43, 39]. The general idea of minimal distortion is to penalize any generalized or suppressed value. When working with generalized values, ILoss [47] is a metric designed the information loss. The discernibility metric from [44] charges a penalty to each record which is indistinguishable from other records with its quasi-identifiers. This metric works exactly against k-anonymity.

When the purpose of the data is known, the publisher can design specific metrics to measure the quality of the published data. If a data set is released for a data mining task, the classification errors could be used. In general, a metric to measure an anonymization step is to measure the information loss in relation to the increase of privacy.

If any and which of these techniques apply in this thesis depends on the chosen anonymization technique and on what needs to measured.

2.3 Conclusion

From the literature we identified the scope and purpose of logging. Challenges of logging were discovered and we decided to partly mitigate them by using an existing log storage technology. For anonymization, an overview of the available techniques, attack models and measurements is provided. Which parts of these three are relevant for this thesis will be researched in Section 5.

3 Requirements

This section discusses the requirements for both logging and anonymization. In both cases, the requirements do not only come from *Het NIC* and literature, but from developers themselves as well. Combining these will clarify the requirements in order to design the logging and anonymization systems.

3.1 Log component

As described in Section 2.1, literature gives insights on how logs are being used. Such as error debugging, system monitoring, anomaly detection and user activity logging. However, we are solving a specific problem, namely logging more data for solving bugs which cannot be reproduced on developer environments. Developers will be the primary users of the logs and therefore we need to know what developers need from logs. To discover what developers need from logs, interviews were held to discover how they think about logging in combination with privacy sensitive information. This section first discusses the interviews, uses the interview results to make the requirements and thereafter lists additional requirements both from *Het NIC* and from the challenges as described in Section 2.1.

3.1.1 Developer interviews

The main goal of the interviews is to discover which (privacy sensitive) information developers need from logs when localizing and reproducing bugs. The interviews were held in a semi-structured way. We constructed a predefined list of questions, but more questions were asked depending on the answers of the interviewee. The questions were designed in such a way to retrieve the experience of the developer as well as let them be creative and think of their ideal logging situation. The predefined questions were as follows:

1. Can you shortly explain what your experience is with software engineering?
2. Do you have experience with bugs that occur during running the application in a production environment and can you give examples?
3. For each example the developer gives:
    - Which information about the bug was available in the logs?
    - Which steps did you take to locate and solve the bug? What was the method you used?
    - (If not answered by previous question): Which role did the logs have in solving the bug?
    - (If not answered by previous question): Which role did data records have in solving the bug?
    - (If not answered by previous question): Was there any privacy sensitive data, such as customer names, addresses or financial records, involved when locating and solving this bug? If so, would you have been able to locate and solve the bug if this privacy-sensitive data was unavailable?
    - Was there information you needed or would have liked to have when solving the bug but which was not available to you?

4. Picture this: there is a bug on the production environment which cannot be reproduced on develop or test environments. This means that you need to look at the logs of the production environment. Which information would you like to see in these logs to help localize and reproduce the bug? You can think of anything you like.

5. Conclusion: is there something you would like to add of which you think can be of help?

Interview results

Seven developers with different backgrounds were interviewed. Two were developers at *Het NIC* who have experience with APRA, the main product the logging solution will be developed for. Three developers work at a company with a larger user base which generates about 100 gigabytes of log events per day. The other two developers both have 3 years of experience at different companies. The full interview reports can be found in Appendix A. To summarize the results, we extracted features from the developers' answers by looking at what kind of information they found useful or would like to have. By exploring both experience and ideal logging situations of the developers, we discovered that most developers mentioned similar features and there were only a few uniquely mentioned features. The following features are in order of number of mentions, from high to low.

System metrics: System metrics are a way to monitor the health and performance of a system at any given time, such as CPU usage, RAM usage and request durations. Mostly these metrics are used to monitor performance in real time and up or down scale the system when needed. Developers are interested in the performance history when attempting to find the cause of a bug. They will look at the metrics at the time an error occurred to see if there were any anomalies. For example, there could be an unusual high number of requests at the moment an error occurred, which could give an indication to its cause.

Log related data records: The second most mentioned feature was the ability to see to which data records an error relates. Since most interviewed developers have access to the production database, they were also satisfied if there are identifiers of the related data records available. Developers value the visibility of related data records, since many of their bugs are caused by malformed data. If they have access to the related data record, they can determine the cause of the error much faster.

Log exceptions with stack traces: Stack traces are an important tool for developers when localizing bugs. They are often recorded at the moment of an unhandled exception. A stack trace allows tracking the sequence of nested functions called, up to the point where the stack trace is generated. That point is the line of code which caused the exception. Next to the stack trace is often the type of exception, which help developers in determining the cause of the error.

Trace user activity/path: When there is a bug that is caused by invalid data, the developers are interested in the steps a user took before the bug occurred. There is a chance that a previous step in the user's process caused the insertion of the invalid data. To fix such a bug, the developers need to alter code in a step earlier in the process than the location of the error. Having the ability to trace user activity, i.e. follow the path the user took through the application, will help the developers in determining which code caused the malformed data.

Query logs: If there are log files with thousands of log events, it becomes difficult for a developer to find a specific log event. The ability to query logs and quickly filter out log events developers are not interested in, will save costly time.

Audit trail logging: The developers' definition of an audit trail is a database level change log. This audit trail is a chronological collection of all create, update, delete and optionally read queries to show how data records evolve over time. An audit trail is a form of activity logging, which reveals which queries which users performed on the database. Developers use it to reproduce erroneous queries or discover how invalid data entered a system.

Log what a user has seen: Logging what a user has seen is an extension to logging the data that is communicated between server and client. This logging could be performed by taking screen shots or reproducing a view with the help of a log event. Seeing what a user has seen makes it easier for a developer to determine what exactly went wrong for the user and is especially useful for front end errors.

Log HTTP Request Header information: The HTTP Request Header contains information like URL and its parameters, HTTP response code, HTTP method, etc. Especially the URL and its parameters are important to localize a bug in the code, because those reveal which method was called by the user.

Log user information: Logging user information is important. Logging the minimal user information, the user ID, is needed to properly query the log events, as well as tracing the activity of a user.

Log data that is communicated between server and client: As an extension to tracing user activity, developers would like to see the data that is send and received by a user. Developers can use this data to quickly localize, reproduce and debug a bug, since the information contains the values of variables at the moment a bug occurred.

Color code/highlight log events: Querying the logs can still result in a large amount of log events. For example, if a log event with an exception is highlighted with a red color, developers can recognize them easily in the large list of log events.

Decrease developer effort to make a log statement: Writing a log statement should take low effort. If writing a log statement takes more effort, developers are less likely to write them which results in less information available in the logs.

Virtual machine dump: Dumping the complete state of the virtual machine when an error occurs gives the developers the possibility to load that virtual machine into their own local environment. Then they can debug the steps up until the error and see the values of all variables, which makes debugging of a live environment possible.

Log front end errors: In a web application, errors do not solely occur server side. The front end can also throw exceptions, often logged in console of the browser. It is not standard to record these exceptions, but they can be a great help for the developer when a bug is front end only.

Log the path of code through multiple services: Complex systems can have multiple services with different responsibilities. When an exception occurs in a service, only the stack trace of the current is known. Tracking the code execution paths between services could help developers in locating and finding the cause of bugs.

Summary of logs around time of exception: This summary could be a report of user activity and system performance around the time a bug occurred. These metrics can give a developer hints as to where to look for the cause of a bug.

Log application version: When software is running at different locations, versions between them can differ. A release could be released for one environment but not yet for another. The difference in application versions means that bugs are fixed on one environment but still present on another. When a developer is asked to look at an error in a specific environment, it is possible to check which version the application is, such that a bug is not solved twice.

Log environment: An environment is the name of a server or a machine. Recording which environment a log event is written by can help developers when searching for the cause of the bug, as well as pointing them to which system metrics of which machine they should look. Logging the environment is only necessary when each server logs to a central place.

Table 1 shows each feature with how many times they were mentioned and their given key.

Table 1: The features of logging as wanted by developers

#	**Feature**	**Times mentioned**	**Key**
1	System metrics	6	METRICS
2	Log related data records	5	DATA1
3	Log exceptions with stack traces	4	EXCEPTIONS1
4	Trace user activity/path	4	TRACE1
5	Query logs	3	QUERY1
6	Audit trail logging	3	AUDIT
7	What the user has seen	3	TRACE2
8	Log HTTP Request Header information	2	INFO1
9	Log user information	2	INFO2
10	Log data that is communicated between server and client	2	DATA2
11	Color code/highlight the logs	2	QUERY2
12	Decrease developer effort to make a log	2	EFFORT
13	Virtual machine dump	2	VM
14	Log front end errors	2	EXCEPTIONS2
15	Log the path of code through multiple services	1	TRACE3
16	Summary of logs around time of exception	1	QUERY3
17	Log application version	1	INFO3
18	Log environment	1	INFO4

3.1.2 Log component requirements

Now that it is clear what information developers use and want from logs, the requirements can be formed. Some of the features the developers mentioned can be directly translated into a requirement, while others need some more context. This section will define the requirements and mark them with a unique key: LREQ (logging requirement). As stated in Section 2.1, the logging system must at least include error logs and user activity logs. The former will be realized by logging all uncaught exceptions in the web application (LREQ1) and the latter by logging all user requests (LREQ2).

Features 2, 3, 4, 5, 7, 8, 9, 10, 17 and 18 will be part of this graduation. The remaining features 1, 6 and 11 up until 16 will be excluded. Those items are disregarded for various reasons:

- **1, METRICS**: System metrics do not help in answering any of the problems this project addresses. While metrics can give insights in the system health and performance at the time a bug occurred, it is unrelated to logging extra data to help developers. Furthermore, *Het NIC* already uses Application Insights for system monitoring [3].

- **6, AUDIT**: Currently, the applications developed by *Het NIC* have an audit trail in place which saves all database transactions because of the need to restore back ups of the database. These transaction logs can serve as an audit log. The only downside would be that developers would need to correlate a transaction in the audit log with a log of a request themselves. Considering the effort it would take to be able to automatically correlate them, automatic correlation will not be implemented in this graduation project.

- **11, QUERY2**: Color coding logs is heavily dependent on which front end will be used to visualize logs. A good front end which supports extensive log query functionality is more important and it remains to be seen if the front end can be extended with color coding.

- **12, EFFORT**: Decreasing the effort a developers needs to make a manual log does not directly provide the developers with more information to solve bugs. Furthermore, the web applications of *Het NIC* all use logging libraries which are designed to make writing a log statement as easy as possible. One could argue that automatic logging indirectly decreases the developers' effort since they need to write less manual log statements. However, this requirements specifically is about decreasing the effort to write a manual log.

- **13, VM**: While the virtual machine dump is the best option to reproduce bugs on a live environment, developing this feature is a research on its own.

- **14, EXCEPTIONS2**: Logging front end errors is a relatively new concept and would take lots of time to develop since libraries for this function are sparse.

- **15, TRACE3**: This feature is a research on its own, because relating code execution paths through multiple services with each other is difficult. Furthermore, *Het NIC* does not have applications consisting of multiple services yet.

- **16, QUERY3**: Similar to QUERY2, this feature is also dependent on which front end is chosen. This feature would also need extra research as to what needs to be summarized exactly.

Features 2, 3, 4, 5, 7, 8, 9, 10, 17 and 18 will be transformed to requirements in the following way:

- **2, DATA1 + 10, DATA2**: The data records that can be directly related to a request are inside the data that is returned by the server when the user executes a request, in other words the data a user has seen on their screen. The server can return web pages (views), JSON data, files, errors, etc. Furthermore, a user can also send POST requests where data is included, which also counts as data a user has seen. Therefore, the data that is sent by the user and returned by the server for a request will be logged (LREQ3).

- **3, EXCEPTIONS1**: This feature is already captured by LREQ1, which states that all uncaught exceptions will be logged.

- **4, TRACE1 + 7, TRACE2**: To solve a bug, the developers often read log events to understand which steps a user has taken to come to the bug. When they have access to the data related to requests as described by LREQ3, they can already track the steps of the users in more detail. However, the ability to actually see what the user has done and seen will

further quicken and enhance their understanding of a bug. Therefore, developers will be given the ability to select a log event to see a reproduction of the screen the way the user saw the view at the moment of the log event. Furthermore, the developers can scroll chronologically through the logs to simulate the steps the user has taken. Thus, LREQ4 is the ability to replay log events.

- **5, QUERY1**: Developers can query and filter logs efficiently via a GUI (LREQ5). This requirement can be realized by either developing a front end to query logs ourselves or use an existing package.
- **8, INFO1**: All relevant HTTP Header information will be logged for a request (LREQ6).
- **9, INFO2**: The user ID will be logged for a request (LREQ7). The user name will not be recorded since that is privacy sensitive and not relevant for the developers.
- **17, INFO3**: Having the application version in a log will help the developers determining whether a bug has been fixed for the version the application is running in. So, the application version will be logged for a request (LREQ8).
- **18, INFO4**: For similar reasons as in LREQ8, the environment (e.g. production, test) will be logged for a request (LREQ9).

Thus, we now have nine requirements for logging from the developer interviews.

Additional requirements

Het NIC aims to use the new logging system in their production environments. This gives some additional requirements that do not come from the developers but from *Het NIC*. The following additional technical requirements must be implemented as well:

- **Network:**
  - Locally store logs before sending them to have a back up in case of connection loss (LREQ10).
  - The server must use HTTPS (LREQ11).
- Logs from developer environments must be saved locally (LREQ12).
- Existing log configuration must also log to the new log component. There must not be more than one location for logging and developers must have one uniform way of logging (LREQ13).
- The web application side of the new logging system must be library packaged as a NuGet package so it can be included in different projects (LREQ14).
- There must be a retention and back up strategy in place (LREQ15).
- There must be a single script to completely set up an empty logging server, which installs all software and prepares all the needed settings (LREQ16).
- There must be an access control mechanism (LREQ17).

- There must be compression in place for the returned and post data to save on storage space requirements (LREQ18).
- Developers accessing logs should be logged in an access log (LREQ19). This will provide information in case sensitive information is released via the logs.

Furthermore, from the problem description and the challenges as described in Section 2.1 follow some requirements as well:

- The developers should not have to write the request logging for every URL in their web application, because there is the risk that they forget to include it or use a wrong formatting. Every request, including its data, should be automatically intercepted and logged (LREQ20). LREQ20 requirement differs from LREQ2, since LREQ20 requirement states that it must happen automatically and not require any manual log statements from developers.
- Extracting all information in a request will take a small amount of time, nevertheless, the user experience should not be negatively affected by introducing extended logging. Therefore, logging a request including the data should not negatively affect the performance of the web application (LREQ21).
- Logging all data related to a request implies a much higher logging volume, therefore the log storage system should be able to handle large volumes of logs (LREQ22).

In total, there are 22 requirements for logging, nine of which come from the needs of the developers.

3.2 Anonymization

A first step in developing the requirements for anonymization is discovering which features of data the developers are interested in when reproducing and fixing bugs. Literature about this specific subject is very sparse. Therefore, a group discussion is held with the developers from *Het NIC* to discover which features of data they are interested in. The wishes from the developers that result from this group discussion are analyzed for their feasibility. Then, the requirements follow from the developers' wishes and the analysis.

3.2.1 Group discussion

The group discussion is setup as follows. A list of all possible types (e.g. string, boolean, integer, etc) of information that can occur in logs of APRA is constructed. Each type is accompanied with an example, such that the developers can directly see what is meant with the type and discuss which information from that type they need to preserve to help when solving bugs. To lead the discussion, a set of basic questions is prepared around these data types. The questions are kept basic such that they start a discussion and not steer the developers in a certain direction. These questions were accompanied with examples of data that illustrated all the different data types. Table 2 shows the data types and their corresponding questions. Note that all types of numbers are combined into one set of questions, since it is assumed that different type of numbers are difficult to distinguish.

Table 2: Data types and questions

Type	Subtype	Questions
-	-	1. Should data types be preserved (e.g. string is anonymized to another string)?
String	Unstructured text	2. What should be done with non-alphanumeric characters in strings? 3. Is there a difference between the anonymization of named entities and unstructured text? 4. Should the structure of the string be preserved (length, spaces, casing, separators)?
	Named entities	See question 3
	HTML/XML	5. What should be done with HTML/XML during anonymization?
	Base64	Base64 strings can represent any sequence of bytes. Therefore, no questions are asked since base64 is privacy sensitive by default
	NULL/undefined	No questions since this is similar to NULL
	Empty	No questions since this is similar to NULL
	JSON	6. What should be done with JSON during anonymization?
	Date and timestamp	7. What should be done with dates during anonymization?
	Filenames	8. What should be done with filenames during anonymization?
Integers	IDs	9. What should be done with numbers during anonymization? 10. Should negative/0/positive values be preserved? 11. Are outliers of any interest?
	Enum values	-
	Counts	-
	IBANs and Personal identification numbers	-
Floats	Percentages	-
	Amounts	-
Booleans	-	12. Can booleans be privacy sensitive and how should they be anonymized?
NULLs	-	13. Should NULL values be preserved?

During the discussion, the prepared questions were answered and some additional suggestions were made.

1. Should data types be preserved (e.g. string is anonymized to another string)? The developers were unanimous, data types should definitely be preserved. A client sending a wrong data type can be the cause of an error, so having the original data type will help when debugging an error while altering the data types will make the logs less useful. Furthermore, from an anonymization viewpoint, it makes no sense to change data types, since there is no privacy gain

(e.g. transforming integer 2 into string "2" does not help in preserving any kind of privacy sensitive information).

2. What should be done with non-alphanumeric characters in strings? One of the examples had a long string in the data with characters such as ß or %&$. The developers agreed that the characters from languages with other alphabets should be anonymized since they can also contain sensitive information. Bugs related to non-Roman characters are becoming increasingly rare since current applications can handle them out of the box. The developers reasoned that non-alphanumeric characters should be preserved, since they sometimes have special meanings in strings.

3. Is there a difference between the anonymization of named entities and unstructured text? The developers do not think there should be a difference between the anonymization of named entities and unstructured text. Firstly, there would be the problem of detecting named entities and secondly, different anonymization rules could disclose more information than intended by accident. Lastly, unstructured text can contain named entities in them.

4. Should the structure of the string be preserved (length, spaces, casing, separators)? Since separators are non-alphanumeric, they will be preserved according to the previous question. The developers thought that it was a good idea to preserve the length, spaces, new lines and tabs inside text. On the other hand, they did not think that preserving casing would be of any added value.

5. What should be done with HTML/XML during anonymization? There was an example where a log contained an HTML string. The developers agreed that the first thing that should be done is anonymizing all normal text contained within the HTML. Furthermore, HTML-tags and their attributes are not privacy sensitive by default, however someone can still include URLs, Base64 data and possibly other information inside attributes. Therefore, the structure can be preserved but all attributes and text should be anonymized.

6. What should be done with JSON during anonymization? An example was taken from a real system log, where there was a JSON string in the log data. The developers came to a quick conclusion: the anonymization tool should work on JSON data, so then the tool should decode the JSON and recursively anonymize that JSON and replace the existing JSON string with a string representation of the anonymized JSON.

7. What should be done with dates during anonymization? After discussing for some time, the developers concluded that dates can be quasi-identifiers. One person noted that there is a difference with standalone dates and dates inside text, while the two are both represented by strings when dealing with JSON. Then they agreed that standalone dates should be preserved, since they are parsed server side and could cause bugs, while dates inside text do not need to be preserved as they are not parsed and therefore not sensitive to faults.

8. What should be done with filenames during anonymization? In some parts of the system, users upload files and the filename they upload is preserved, which means that a filename can occur in log data. A filename can include privacy sensitive data such as a company name. The developers agreed that the same anonymization method as for normal strings should be used for this, but that the file extension should be preserved, because a file extension is not privacy sensitive while it can give information about the cause of an error.

9. What should be done with numbers during anonymization? The developers were considering the different uses of a number. When a number is used as an ID for an object, it is not privacy sensitive since the developers have no access to the production database. Even if a developer has access to that production database, using IDs from anonymized logs is probably the

least of the company's problems since they actually have a data leak (a developer having access to the production database) at that point. Numbers are also used for enum types. An enum in C# is represented as an integer in JSON. The developers agreed it would be useful to see the real enum values, since they are often used in if-statements and queries. For the rest of the uses, personal numbers, amounts, percentages, etc, the developers thought it would be best to randomize them while preserving the length of the number.

10. Should negative/0/positive values be preserved? The developers agreed that negative numbers should stay negative and positive numbers should stay positive during anonymization. The chance that some value is positive or negative is privacy sensitive is small and could help in identifying an error. For example, a negative number for an ID indicates something is wrong with that field. Furthermore, the developers would like that 0 stays 0.

11. Are outliers of any interest? Before thinking if outliers are useful, the developers concluded that it would not be possible to detect outliers reliably. For example, the anonymization tool cannot know if a date with value 2100-01-01 is an outlier for a particular field. The developers agreed that outliers could be an indication of an error, but not anonymizing them would be vulnerable to privacy leaks.

12. Can booleans be privacy sensitive and how should they be anonymized? The developers were not able to think of a situation where a standalone boolean could be used to identify something. However, a boolean can be a quasi-identifier. The discussion was continued with the question: so if a boolean is a quasi-identifier, should it be anonymized? The developers reasoned that, firstly, booleans are important to know when reproducing a bug since they often determine the code paths. Secondly, when anonymizing a boolean value, there is no method which does so reliably. For example. swapping a boolean value will not help in anonymizing. Randomly choosing either NULL, true or false still has a $\frac{1}{3}$ chance of being the original value, but has as benefit that it is impossible to discover the original value since that is included as well.

13. Should NULL values be preserved? The developers were unanimous, NULL values should be preserved. This is no problem, since having no value cannot be used to identify something. It could however disclose some information when the distribution is skewed (e.g. 1 in 100 values is NULL). One developer opted to include special strings as well. Strings with values "null" and "undefined" (JavaScript notation of NULL) should not be anonymized and stay as they are. This should be done since a string can contain these words when an object with no value is concatenated with a string. If these words are removed, a developer can no longer see which variables were NULL.

Next to the answers, the developers came up with some interesting propositions.

Integers will be anonymized, which means that enums will lose their original value. However, enums are not privacy sensitive by themselves, as enums always denote some type of category and not a concrete entity. So it would be nice if the JSON-parser could detect an enum and write some extra information in the log to tell the anonymizer that it is not needed to anonymize that particular value.

Similarly, another developer mentioned the use of attributes. For example, there is an attribute that is often used in classes, named MaxValue, which can be used by programmers to easily add a max-value-constraint on number fields in order to not have to write an if-statement which checks the numbers value. This attribute helps with code readability and code duplication. The developer said that it would be nice if the anonymizer could take such attributes into account. For example: an integer field has a MaxValue attribute of 50. Then if the anonymizer encounters that field and discovers that the value is actually 60, then it should not be randomized to a value lower than 51,

since that could be the cause of the error. Vice versa, the anonymizer should not randomize values lower than 50 to values greater than 50.

Lastly, another developer thought of an edge case. In JSON, it does not matter if a number is larger than the maximum value of an integer since integers and longs are one and the same for JSON. While integers and longs are represented by different data types in the C# web application. So, when a number is encountered which is larger than the maximum integer value, it should not be randomized to a value which is less than the maximum integer value as that could be the cause of an error.

3.2.2 Anonymizer requirements

To summarize the results from the group discussion, Table 3 contains the developers wishes for anonymization. These wishes can be mostly directly translated into requirements, however, not every developers' wish is realistic. Some of the wishes, if implemented, will have a negative impact on the privacy or are unfeasible.

Table 3: Developer anonymization wishes

Feature	**Key**
Data types should be preserved	DATATYPES
Booleans do not need to be anonymized	BOOLEANS
NULL values should be preserved	NULLS
Strings containing 'null' or 'undefined' should be preserved	STRINGNULLS
Randomize numbers while preserving their length	NUMBERS1
Preserve ID and enum numbers	NUMBERS2
Negative numbers should stay negative and positive numbers should stay positive	NUMBERS3
A number with value 0 should stay 0	NUMBERS4
Non-alphanumeric characters should be preserved	STRINGS1
Length of strings should be preserved	STRINGS2
White space in strings should be preserved (spaces, tabs, enters)	STRINGS3
Standalone dates should be preserved	DATES
HTML should be preserved, but text and attributes should be anonymized	HTML
JSON strings should be anonymized recursively	JSON
Extensions of filenames should be preserved	FILENAMES
Include property attributes in the log data	ATTRIBUTES
Numbers larger than integer min/max value should not be anonymized to a value lower than integer min/max value	NUMBERS5

The following wishes from Table 3 will not be implemented due to privacy reasons:

- **NUMBERS2**: IDs of objects are not privacy sensitive. Most enums will not be privacy sensitive, but some will be. Regardless of whether those fields are privacy sensitive or not, detecting whether a number is an ID or an enum is a problem on its own. While it would be possible, as the developers suggested, to record which fields are enums in log data, such technique is not possible for IDs since they are just denoted as integers. Furthermore, such an enum detection method will cost much time to implement. Name recognition could be used to detect fields with "ID" in their names, but that will not cover all fields that are IDs or give

false positives. Therefore, both IDs and enums will be anonymized according to the standard number anonymization rules.

- **NUMBERS4**: The developers reasoned that numbers that have a value of 0 cannot be privacy sensitive. However, they also concluded that amounts and percentages should be anonymized. So they contradict themselves when a percentage or amount is 0. Therefore, numbers that have a value of 0 will be anonymized according to the standard number anonymization rules.

- **DATES**: Standalone dates should be preserved according to the developers. However, dates can be strong quasi-identifiers, e.g. date of birth, which means that standalone dates should be anonymized.

- **FILENAMES**: The developers would like to see the extension (e.g. .pdf, .xls, .txt) of any filenames that occur in strings. While these extensions will never be privacy sensitive, it is unfeasible to implement a method to correctly recognize all filenames. Such an algorithm also has the risk of false positives by marking privacy sensitive text as filenames. Therefore, in regard of privacy, there will not be a detection algorithm for filenames and thus will be anonymized according to the standard string anonymization rules.

- **ATTRIBUTES**: Automatic detection of attributes of properties such that, for example, the anonymizer knows the valid range of a number, will enhance the utility of anonymized log data. However, similar to enums, implementing this feature will cost much time and will therefore not be implemented.

The following requirements possibly have a negative impact on privacy and need some form of risk analysis:

- **STRINGS1, STRINGS2, STRINGS3**: Preserving string length plus all white space and non-alphanumeric characters in strings can be a privacy risk when a creditor can be identified by a unique combination of length and positions of these preserved characters. A risk assessment will be conducted to determine the privacy impact of preserving string length and these characters.

- **BOOLEANS**: Whether to disclose a boolean depends on its distribution. For even distributions, the boolean can be disclosed safely. However, it is not always possible to reliably determine a distribution during anonymization, since log data for a certain model or view can be scarce. Therefore, two options remain: randomize the boolean value or disclose the original value. Naturally, a randomized value does have a negative impact on utility while disclosing the original value can possibly have an impact on privacy, but only if booleans on themselves can be privacy sensitive. A risk assessment will be conducted to determine the privacy impact of disclosing booleans.

- **NULLS**: Similar to booleans, the developers think that NULL values cannot be privacy sensitive. However, it could be possible that a NULL value does give away some information when there is an uneven distribution, e.g. 1 in 100 values is NULL. There will also be a risk assessment conducted for NULL values.

Next to the developers wishes, there are two more features that will be included. Firstly, a part of URLs can be easily detected and preserved, the URI schemes [12] which are words like http, mailto and callto. This feature will help developers as they can see that there was a URL in a string. Secondly, fields in the log data with the same path and same value will need to be anonymized to the same value such that developers know that the original values were equal. For example, the log data contains an array of invoices. Each invoice has a field called CompanyName. Equal values in the field CompanyName will be anonymized to the same value such that the developers know which invoices belong to the same company and how many unique companies there are in the log data.

Dynamic anonymization

When inspecting log data, developers will often have a hunch as to which field is the cause of an error. The developers also mentioned that they were interested in the fact if the value of a number or date is within or outside a certain range. However, as stated before, those ranges will not be included automatically since the ATTRIBUTES requirement will not be implemented. Instead of automatic inclusion of ranges, the developers should be able to ask whether a certain field was within a range. This way, developers can confirm their hunches and better utilize the log data. Naturally, this feature will have an impact on the privacy. The details, implementation and privacy risks of this dynamic anonymization will be discussed in Section 5.

Requirements list

The anonymizer has three general requirements and Table 4 shows the data type specific requirements. Similar to logging requirements, each anonymization requirement (AREQ) will be given a key.

- Data types should be preserved (AREQ1)
- Fields with the equal path and value should be anonymized to the same value (AREQ2)
- Dynamic anonymization (AREQ3)

Table 4: Data type specific anonymization requirements

Type	**Subtype**	**Description**	**ID**
String	Unstructured text	Standard anonymization	AREQ4
	Named entities	Standard anonymization	AREQ5
	HTML/XML	Preserve structure but anonymize text content and attributes	AREQ6
	Base64	Preserve type but anonymize	AREQ7
	NULL/undefined	Preserve, same as NULLs	AREQ8
	Empty	Preserve, same as NULLs	AREQ9
	JSON	Anonymize recursively	AREQ10
	Date and timestamp	Same as numbers	AREQ11
	Filenames	Standard anonymization	AREQ12
	Numbers	Same as numbers	AREQ13
	-	Preserve length	AREQ14
	-	Preserve white space	AREQ15
	-	Preserve non-alphanumeric characters	AREQ16
Numbers	-	Standard anonymization	AREQ17
	-	Preserve negativity/positivity of numbers	AREQ18
	-	Preserve number length	AREQ19
Booleans	-	Preserve	AREQ20
NULLs	-	Preserve	AREQ21

3.3 Conclusion

In total there are 22 requirements for logging, discovered with the help of the interviews with developers and technical requirements from *Het NIC*. The group discussion held around the subject of log anonymization revealed 21 requirements for anonymization. The next logical step is to realize the logging requirements. This will be done first since the anonymization needs to be built in the new logging system.

4 Implementation of log component

This section discusses the implementation details of the log server and log library. Inside the implementation description, each requirement as discussed in Section 3.1 will be marked when addressed. First, the overall system architecture will be discussed. As a second step, the log storage will be realized in detail. Thereafter, the web application library will be discussed, including the design of capturing web requests and their data. Lastly, the requirement that defines the replay functionality (LREQ4) will be discussed separately due to its complexity.

4.1 System architecture

An overview of the architecture of the logging system can be seen in Figure 2.

Figure 2: Logging system architecture overview

[Figure redacted for publication]

ElasticSearch and Kibana are chosen for respectively the log storage and the log querying. Their implementation details will be explained in Section 4.2. Kibana has a connection with ElasticSearch and is configured in such a way that only the non-privacy sensitive log events are visible and searchable. An important design decision is to prevent any outside access to ElasticSearch and insert a proxy between the web application and ElasticSearch. This is done for two reasons. Firstly, this proxy is the only access point to privacy sensitive log data and is programmed in-house, which means that we have full control over the log data that leaves the Ubuntu server. Secondly, the intuition was to build a plugin for ElasticSearch that anonymized every piece of log data whenever one gets requested. However, currently, ElasticSearch has a limit of 1 plugin hooking into requests. Since SearchGuard is installed and hooks into requests to enforce HTTPS, it is not possible to build the anonymization method as a plugin for ElasticSearch.

Therefore, the .NET Core proxy was designed instead, which performs on-the-fly anonymization every time log data gets requested. The web application only communicates with this proxy over HTTPS and has no direct connection to ElasticSearch. Combined with access control in Kibana, it is guaranteed that log data does not leave the ElasticSearch server in its raw form. In turn, the .NET Core proxy logs whenever log data is accessed (LREQ19).

Developers can use Kibana to search for log events, which are logs of requests and the details will be explained in Section 4.2. These log events do not contain privacy sensitive information related to the requests. The log data, which can be accessed via a web page in the log library of the web application, contains the privacy sensitive data related to a request. As can be seen in Figure 2, when developers request to view log data, the log library issues a request to the .NET Core proxy, which in turn requests the log data from ElasticSearch, applies anonymization to the log data and then returns it to the log library. The developers can then view the log data in raw format or can choose to recreate the view as the user saw it with the help of the log data. The details of the replaying functionality will be explained in Section 4.4.

The complete Ubuntu server can be initialized from an empty Ubuntu installation by running a single script (LREQ16). This script also registers a periodic job on the server which deletes log events and log data older than X days (LREQ15).

4.2 Log storage

Instead of writing custom log storage, with all its challenges, the log events and log data are stored in ElasticSearch [5]. ElasticSearch is a free system designed specifically for the digestion of a high volume of logs while maintaining very low latency (LREQ22). It features an extensive query language to efficiently search through events and supports plugins. ElasticSearch uses so called indices to store data in JSON format, which can be roughly compared to a table in a database. The back end of ElasticSearch is Lucene [2], which automatically finds optimal compression for logs (LREQ18).

Kibana [5] gives developers the ability to search through the log events (LREQ5). It functions as a front end for ElasticSearch and is developed by the same company. Next to querying, Kibana can also generate different kind of statistics about log events, e.g. a graph of amount of exceptions per time frame. Kibana and ElasticSearch do not offer access control in their free versions. However, the plugin Search Guard [10] has a free version which will be used to hide certain indices for certain users and protect both ElasticSearch and Kibana with HTTPS (LREQ11).

An important design decision is to store log events and their data separately, in different entities.

Log event: A log event is a log of a request and contains the following information (LREQ6, LREQ7, LREQ8, LREQ9):

- Implementation details redacted for publication

Log data: Log data is linked to a log event via a foreign key. Log data contains data posted by the user and data returned by the server for a request.

Separation of log events and log data means that there is a LogEvent index and a LogData index. The LogEvent index has a predefined schema while the LogData index is flexible, meaning it can store any structure of data. Furthermore, search optimization is disabled for the LogData index. Users will never search through log data directly and disabling search optimization speeds up the insertion time of log data.

Storing them separately gives freedom in access control (LREQ17). In the standard configuration, developers are able to access and search the log events but cannot access the log data freely. To access log data, a developer must request access via the ID of a log event. That way the system has full control over which data is released with what kind of anonymization.

Furthermore, the separation also enables the possibility of physically separating the log data from the log events. One could choose to store the log data on a server with stricter access control rules and better security, for example.

Both indices are not meant to store log statements that the developers manually inserted in the code (LREQ13). A separate index should be created for that and the schema of that index is dependent on what needs to be logged around a message. For *Het NIC* the schema exists of a timestamp, log level (info, debug, warning, error, fatal), controller and message. Then the existing log technology should be adjusted such that it sends it logs to the ElasticSearch server. For example, APRA uses NLog [8] which has so called log targets and can be extended with a log target to send developers' logs to the .NET Core proxy.

4.3 Logging library for the web application

Recording and sending the log events and their data to ElasticSearch is done via a custom built library (LREQ14). The library is fully configurable, can be included in any ASP.NET web application project and only needs around 30 lines of code to work. The library offers two methods to record logs, a method to process and log a normal request (LREQ2) and a method to log a failed request that ended in an uncaught exception (LREQ1). The signature of the methods is as follows:

```
Log(Request, Result, Timestamp, UserID)
LogException(Request, Exception, Timestamp, UserID)
```

The `Request` object contains all information related to the request, such as URL, parameters, status code, etc. The `Result` object contains the result of the request, which can be a view, JSON, file, etc (LREQ3). [Implementation details redacted for publication] The `Log` method then automatically captures all requests and developers do not have to write any log statements themselves (LREQ20). Note that the developers are still free to put any other log statements in their code (LREQ13). [Implementation details redacted for publication] Inside this method, `LogException` can be called such that every failed request is recorded. Again, developers do not have to write any log statements themselves. When one of the two methods is called, the library transforms a request and its results into a log event with log data, using JSON serialization. This serialization is done in a different thread such that performance is not impacted and the user does not experience any delays (LREQ21). The log event with log data is then sent to ElasticSearch via the .NET Core proxy. Table 5 shows what data is serialized per result type. The POST data is serialized for all results.

Table 5: What data is serialized to log data per request result type
[Implementation details redacted for publication]

When developers want to view log data, they can use the ID of a log event found in Kibana and enter that ID in a special view which is shipped with the log library. The library sends a request to the Ubuntu server and the .NET Core proxy responds with the anonymized log data. Figure 3 shows the view where developers can retrieve log data. As can be seen, they need to enter the ID of the log event and their personal token to retrieve log data. They can also enter rules for dynamic anonymization, which will be explained in detail in Section 5.1. When they press the "Get log data" button, the .NET Core proxy will return the anonymized log data in JSON format. If the developers click on "Replay", the given log event will be replayed.

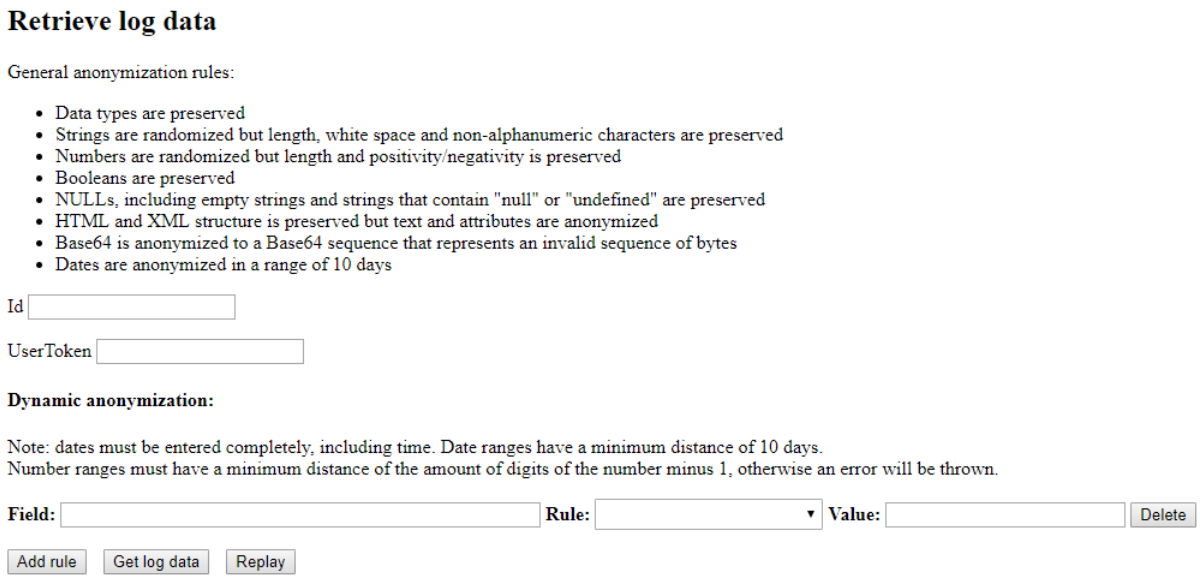


Figure 3: Screenshot of the view where developers can retrieve log data

The library also keeps a local backup of log events and their data for 2 days in case there is a connection loss (LREQ10). Furthermore, there is a setting to write logs locally which should be used by developers of the web application in their development environment (LREQ12). A summary of the features:

- Write log events and exceptions to an ElasticSearch environment
- Write log events and exceptions locally
- Configurable per environment (e.g. develop, test, live)
- Configurable fields to ignore during serialization (e.g. password fields)
- Ability to add custom serializers
- View anonymized log data

4.4 Replay functionality

The idea for replay functionality, a replayer, comes from the developers' wishes to be able to more extensively trace what a user has been doing around the time a bug occurred (LREQ4). A developer can use the replayer by going to a special URL on the environment the replaying should take place. The ID of a log event should be entered on that screen and if that ID belongs to a log event which resulted in a view, the replayer regenerates the view as the user saw it at the time of log event. The view can be rebuilt by using the log data belonging to the log event. Naturally, this log data should be anonymized. The replaying functionality is included in the logging library that is installed in

the web application. In order to understand how the replayer works, some background information on ASP.NET is needed.

[Implementation details redacted for publication]

Figure 4: Sequence diagram of rebuilding a view from a log event

[Figure redacted for publication]

The replaying functionality can been seen in Figure 5. Using the replayer, developers can browse through the different actions of a user. Furthermore, all actions executed on the current page can be viewed by clicking the extend button, which can be found above the log event that is currently being replayed. This button reveals a table with information about those actions. In the example of Figure 5, the page loads with a simple model and all creditor data is retrieved via an AJAX request. Then the user executes some more requests on this page, before moving on to the next view.

Figure 5: Screenshot of replayer

[Figure redacted for publication]

Limitations

Using the replayer in a live environment has some drawbacks. [Implementation details redacted for publication] unintended manipulation of data could occur. Secondly, views contain links and other functionality which can cause an unintended call to the live server, which could result in data manipulation or data leaks. Although the replayer will capture most of these requests, it is unfeasible to capture them all. Thirdly, access control needs to be tightly configured to give developers access to the replayer but not to any other content on the live environment.

To address these three problems, replaying should only be allowed on a special environment which uses a dummy database. Setting up and maintaining such an environment can be time and cost expensive. [Implementation details redacted for publication]

Furthermore, the replayer cannot distinguish between actions performed by users on multiple browser tabs. When an application user used multiple tabs while browsing the web application, the chronological order as seen in Figure 5 is no longer guaranteed to be the correct user flow. Since it is impossible for a server to distinguish requests between different browser tabs, tab information cannot be stored in a log event and therefore this limitation cannot be fixed.

4.5 Conclusion

In a typical use case, when the developer is solving a bug, he would start by searching for the related log events in Kibana. For simple bugs these log events can be enough. When a bug is more complex, the developer can copy the ID of the log event, go to the web page as shown in Figure 3 and paste the ID. Then he can choose to view the raw anonymized log data or use the replayer to reproduce the view as a user saw it at the time of the log event.

We successfully implemented a logging system that automatically records user's activity and what data was visible on their screen. The log storage has high performance with the help of ElasticSearch while the web application library does not have any impact on the response time of the web application. All requirements were addressed and the replaying does work, but with significant limitations. The limitations can be countered by setting up a separate server for replaying, but it remains to be seen if the usefulness of the replayer outweighs such the time and cost effort of maintaining a separate server.

5 Anonymization solution

In Section 4, we described the architecture of the system and the general method for retrieving log data. Kibana is the only access point for log events, while the .NET Core proxy is the only access point for log data. Hence, the anonymization of privacy sensitive log data will take place inside the .NET Core proxy. Next to that, the web application is the only program that will be able to communicate with the proxy. Developers can browse to a web page included in the web application library and enter the ID of a log event they want to see the data from. This data is then retrieved by the web application via the .NET Core proxy. In this section, the anonymization solution will be discussed to anonymize any data as it gets passed through the proxy. The implementation details and algorithms will be elaborated. This anonymization solution tackles the problem of anonymizing data regardless of its structure and data types, while preserving useful information for developers when they are solving bugs. Furthermore, it provides a technique to dynamically anonymize data with user-defined ranges. Similar to Section 4, the requirements will be mentioned during the description of the chosen solution.

The main principle of the algorithm is to randomly permutate everything according to the requirements as defined in Section 3.2 and specifically the rules in Table 4. The other anonymization techniques identified in Section 2.2 do not apply. The main reason is that identifiers and quasi-identifiers can not be defined beforehand, since we need to apply the anonymization on any data structure. It would be possible to implement a function such that developers can mark the identifiers for the log data, but since the developers are the persons who will access the anonymized log data, that would introduce a significant security risk. The reasons why each other anonymization technique does not apply are as follows:

- **Generalization**: Since we do not know the identifiers and quasi-identifiers, everything would need to be generalized. This has a large impact on the utility of the data, let alone define a generalization script for any type and sub type of data.
- **Suppression**: Suppression means removing identifiers and quasi-identifiers from the data, but since the system does not know the fields which are identifying, this technique cannot be applied.
- **Pertubation**: This technique does not apply since we do not have any statistical data nor are we interested in the statistical value of the data.
- **Anatomization**: Anatomization aims to split the (quasi-)identifiers of a dataset and the rest of the dataset into two separate datasets. However, we do not know the (quasi-)identifiers so this technique is disregarded.

Therefore, the chosen technique is permutation. Permutating every piece of data by default is a best practise in anonymization in order to minimize the chance of releasing any privacy sensitive information. The requirements do leave some data types un-permutated with booleans and NULL values or less-permutated in the case of strings and when developers use ranges. Therefore, the privacy risks that come with the chosen anonymization method will be discussed in detail in this section.

5.1 Implementation

To start, we will give a general overview of the anonymization algorithm. Then the anonymization technique of each data type will be described in detail.

Inside the .NET proxy, called the anonymizer, each preservation and conversion rule for each data type is implemented in its own class. Having such a degree of modularity enables the addition of extra rules in the future without introducing breaking changes. The basic anonymization algorithm can be seen in algorithm 1.

Algorithm 1 Recursive anonymization

[Algorithm redacted for publication]

[Algorithm explanation redacted for publication] Users can enter their dynamic anonymization ranges in the view as shown in Figure 3.

Numbers: In JSON, integers and longs are the same. The anonymizer considers every whole number a long to prevent integer overflow errors. Numbers are randomized while their length is preserved (AREQ17, AREQ19), which means that for a 1 digit positive number, the value the anonymizer returns is $[0, 9]$. For every positive number where the number of digits $n > 1$, the anonymizer returns a number in the range $[10^{n-1}, 10^n - 1]$. The sign of a number is also preserved, meaning that positive numbers stay positive and negative numbers stay negative (AREQ18), since this can be important information for developers. Furthermore, 0 is deemed as non-negative, so a single digit negative number is returned by the anonymizer as a value between $[-9, -1]$. For any negative number where the number of digits $n > 1$, the anonymizer returns a number in the range $[-(10^n - 1), -(10^{n-1})]$. These same rules hold when the number is a float. Both the integer and fractional parts of the number are subject to these rules individually, which ensures the length of a float is also preserved.

Considering the dynamic anonymization, users are able to ask whether a number is greater or equal than a certain value, less than a certain value or both (AREQ3). If the user asks whether a number is greater or equal than a certain value, then the algorithm will return a value lower than the user defined value if the original number was less than the user defined value and vice versa. The opposite holds for when a user asks whether a number is less than a certain value. Giving the developers this functionality is not without privacy risks, which will be discussed in Section 5.2.

When users are using both constraints, they are asking the anonymizer whether a number is within a certain range. The algorithm then returns a randomized number inside or outside the defined range, depending on whether the original value lies inside the defined range. The minimum range they can define is as follows. Let n be the number of digits of the original value of the number for which the range is defined, then the minimum range r is:

$$r = \left\{ \begin{array}{ll} 10, & \text{for } 1 \leq n \leq 2 \\ 10^{n-2}, & \text{otherwise} \end{array} \right\}$$

This definition is chosen such that the algorithm does not give too much information with larger numbers. If, for example, the user could enter a range of 10 in a 5-digit salary number, he would be able to pinpoint fairly exact values which is likely just as bad as knowing the original value. In the definition above, the minimum range is 1000 for a 5-digit number, which is much safer. If the user enters a range which is less than the required minimum range they will get an error message.

To illustrate an example, lets assume that the real value of the number that will be anonymized is 6 and see how to anonymization algorithm answers when the user inputs certain ranges in Table 6. As can be seen from the first two anonymized values, the algorithm does reveal which direction of the user input range the original number is. This will give developers some more relevant information to work with if the number is outside the given range. The algorithm pseudocode can be found in Appendix D.1.

Table 6: Example of number anonymization with user defined ranges where the original value is 6, which gets anonymized to a number in a certain range

User input	**Anonymized value**	**Description**
$[100, 200)$	$[0, 99]$	6 is outside the user defined range so anonymize to a value between the default lower limit and the user lower limit minus 1
$[-20, -10)$	$[-9, 9]$	6 is outside the user defined range so anonymize to a value between the user upper limit plus 1 and the default upper limit
$[-5, 5)$	$[5, 14]$	This is a special case where the length of the number is no longer preserved in favor of anonymity. If the length would be preserved, then the range would become too small, so the algorithm increases the anonymized range to at least 10.
$[0, 120)$	$[0, 119]$	Since 6 is in the range given by the user, the algorithm will anonymize to a number in that range.

To give a concrete example of dynamic anonymization being applied, Listing 2 shows some simple data that will be anonymized.

Listing 2: Dynamic anonymization example with original data

```
"Data": [
{
    "Id": 1,
    "Name": "Creditor 1"
},
{
    "Id": 50,
    "Name": "Creditor 2"
}]
```

The JSON path to which dynamic anonymization can be applied is `Data.Id`. Lets say the user wants to know whether `Data.Id` is between 20 and 40, so his query becomes <`Data.Id`, $[20, 40)$ >. Then the anonymizer will anonymize the according to the defined rules, which is shown in Listing 3. As can be seen, the `Data.Id` with value 1 is anonymized to a number in the range $[0, 19]$ and the `Data.Id` with value 50 is anonymized to a number in the range $[40, 99]$.

Listing 3: Dynamic anonymization example anonymized with user defined range `Data.Id` between 20 and 40

```
"Data": [
{
    "Id": 13,
    "Name": "ddd5vZ67 p"
},
{
    "Id": 89,
    "Name": "UpunaChr K"
}]
```

Dates (with times): Inside the anonymizer every date object also has a time, these objects are called date-time objects. If the original date-time value does not have a time component, then the time is deemed 00:00:00. Similar to numbers (AREQ11), date-time objects also have a minimal range of 10, meaning 10 days. For example, the date-time 2019-03-14 10:00:00 will be randomized to a value in the range [2019-03-09 10:00:00, 2019-03-19 10:00:00]. Two special cases are when the value to be anonymized is close to 0000-00-00 00:00:00 or 9999-12-31 23:59:59, the minimum and maximum date-time values, then the algorithm will ensure a minimum range of 10 but not overflow the date-time value.

Furthermore, the user can also apply dynamic anonymization to date-time values and then the same rules as with numbers apply (AREQ3). For numbers, we defined a dynamic minimum range the user can enter, where the lowest is 10. However, for date-time we choose to use a fixed minimum range of 10 days. An alternative would be to use a minimum range for each component of a date-time object, the year, month, day, hour, minute and second component. Then the user could ask whether, for example, the seconds in a date-time object are in between a certain range. However, we choose to use a fixed minimum range of 10 days due to the privacy impact of minimum ranges on components of date-time objects. In many cases, e.g. date of birth, being able to query on an hour, minute or second granularity level will reveal the original date and cause a leak of privacy sensitive information. Therefore, to be safe, we chose to define a fixed minimum range of 10 days.

Strings: Strings have some special cases as described in Section 3.2 in Table 4. When a string does not fall under any of these special cases, its contents are randomized to alphanumeric characters while preserving length, white space and non-alphanumeric characters (AREQ4, AREQ5, AREQ14, AREQ15, AREQ16). The special cases are as follows:

- **HTML/XML (AREQ6)**: Inside strings that are HTML or XML, all normal text is anonymized. Furthermore, every attribute is also anonymized such that no user generated values in attributes are recognizable. This way, full anonymization of HTML and XML strings is guaranteed while the developers can still see the structure of the HTML or XML.
- **Base64 (AREQ7)**: A Base64 string represents bytes. This means that Base64 strings can represent anything and inside APRA they are used for user generated images. Base64 strings must be anonymized for both reasons. They are anonymized by randomizing the contents, while preserving the length, such that it is still a valid Base64 string but the sequence of bytes no longer can be interpreted.
- **NULL/undefined/empty (AREQ8, AREQ9)**: Strings with value ”null”, ”undefined” (NULL in JavaScript) or ”” are all preserved regardless of casing.

- **Dates and timestamps (AREQ11)**: Strings of this type are parsed to date objects, then anonymized according to the date anonymization rules. This means that developers can also apply dynamic anonymization to string dates. The result is then converted back to a string such that the original data type is preserved.
- **Filenames (AREQ12)**: There is no detection built in for filenames, which means that filenames are anonymized according to the standard rules.
- **Numbers (AREQ13)**: Strings of this type are parsed to numbers, then anonymized according to the number anonymization rules. This means that developers can also apply dynamic anonymization to string numbers. The anonymized numbers are thereafter converted back to a string such that the original data type is preserved.

5.2 Risks

The algorithm anonymizes almost every piece of information in log data, but it does disclose boolean and NULL values. By disclosing booleans and NULL values, an attack window is possibly opened for table, record and/or attribute linkage attacks as described in Section 2.2. Since all other information is anonymized, the only thing an attacker can learn is whether a creditor or person is inside the database of *Het NIC*. To assess the risk that comes with disclosing booleans and NULL values, a risk assessment will be performed.

Similarly, the algorithm also preserves string length plus white space and non-alphanumeric characters in strings. This preservation can be a privacy risk when a creditor can be identified by a unique combination of length and positions of these preserved characters. Therefore, a risk assessment will be performed to analyze the privacy impact of preserving string length plus white space and non-alphanumeric characters in strings.

Furthermore, the dynamic anonymization introduces some risks since the users can control the degree of anonymization. By querying the log data often enough it is possible to discover the original value of a number or a date. The attack details and how to prevent such an attack will be discussed in this section.

Lastly, there exists a vulnerability in the web application library regarding deserialization in Json.NET [7], however, this risk can be completely solved.

In these risk assessments, we assume that an attacker has the following intentions: the attacker is a developer who is a legitimate user of the system and tries to discover privacy sensitive information in the logs that he is not able to see by default. Therefore, a risk is deemed as solved or negligible when the possibility to learn privacy sensitive information is non-existent or extremely low. Furthermore, this means that we make a number of security assumptions, such as the log server being secure and that access control in both Kibana and the web application is configured properly.

5.2.1 Boolean and NULL disclosure

Inside the database of APRA, the web application of Het NIC, creditors and persons are represented in many different ways. The booleans and NULL values that are disclosed are only risky when those values can also appear in a third party database, since a combination of those values could possibly identify a unique creditor or person when combined with a third party database. In this risk assessment, the table which contains the invoices will be analyzed in this risk assessment, since that

table is the basis of all data in APRA. Other tables generated by the process of APRA itself will not be analyzed since they are unlikely to be linkable with third party databases.

The most straightforward approach is to discover if any combination of NULL and boolean values disclose a unique or a few creditors. However, the invoices table contains 124 columns, which means that the amount of column combinations is $\sum_{n=1}^{124} \frac{124!}{124!(124-n)!}$, which is unfeasible to analyze.

So the first step could be to reduce the amount of columns by removing redundant columns. Since we are looking for combinations of NULL and booleans values, a column is considered redundant for this analysis when a `NULL`, `true` or `false` value in column A implies the same value in column B. So for every invoice, when column A is `NULL` then column B is also `NULL`. This technique reduces the amount of columns to 39, which is still too much to do a full analysis of all combinations. Therefore, the most common representations of the invoices as they appear in APRA, and thus in the log data, will be analyzed.

There are three common representations, which can be found in Table 7. At no time, all invoices are displayed on a users screen, which means that there will never be log data in which all invoices appear.

Table 7: Three most common representations of invoices
[Representations redacted for publication]

These representations do not contain any boolean columns, which simplifies the queries. However, the analysis could be repeated fairly easily to include boolean columns. The analysis per representation will consist of the following steps:

1. Remove any column that contains 0 NULL-values, since these will never appear in any combination of NULLs.
2. Generate all possible sets of combinations of columns.
3. Execute a query to count the amount of unique creditors identifiable per column combination. An example of such a query: let the set of columns be $(Entity, Address, City)$ then the query is `SELECT COUNT(DISTINCT([CreditorName])) FROM InvoiceWithDatasets WHERE [Entity] IS NULL AND [Address] IS NULL AND [City] IS NULL`.
4. Isolate the column combinations where 1 or a few creditors are identifiable.

Het NIC provided 4 production data sets of different sizes to perform the analysis on. These datasets are of different quality, since they come from different companies. Companies differ in their invoice registration process, resulting in a different amount of information being recorded, which in turn results in a different amount of NULL-values in the dataset. Appendix C shows the full results of the analysis. To summarize, the following combinations of columns with NULL-values reveal a unique or a few creditors:

- **Dataset A, 30.000 invoices**:
  - $(IBAN, Address)$ combination reveals one unique creditor.
  - $(IBAN, InvoiceDescription)$ combination reveals one unique creditor.
  - $(InternalInvoiceNumber)$ and every column combination that includes $InternalInvoiceNumber$ reveals three possible creditors.

- **Dataset B, 170.000 invoices**:
  - $InvoiceDescription$ combined with other columns reveals one unique creditor.
  - $(IBAN, PostalCode)$ reveals two possible creditors.
- **Dataset C, 256.00 invoices**: this dataset has no risks.
- **Dataset D, 439.000 invoices**:
  - $InvoiceDescription$ combined with other columns reveals one unique creditor at worst.
  - $(Address, City)$ reveals eight possible creditors.

For the columns $InternalInvoiceNumber$ and $InvoiceDescription$, the risk is minuscule. Both columns are company specific and therefore unlikely to appear in third party datasets. More realistic would be that an attacker has a dataset where the $IBAN$, $Address$, $Postalcode$ or $City$ columns are included. However, as stated before, each company has a different invoice registration process, making the chance that these exact values are NULL in the attacker's dataset as well extremely small. Besides, there are more reasons the risk is small:

- There will never be log data in which all invoices appear. This means that if the attacker finds a match with a creditor in his dataset, he cannot be certain that it concerns the matching creditor since other logs can contain other creditors which also match that same creditor from his dataset.
- The attacker must know that the creditor he is targeting to reveal is present in the whole database of APRA to be confident of a match.
- The attacker's dataset can contain creditors or persons that matches multiple or different creditors in the database of APRA.

Even if the attacker has a confident match, all the attacker learns is the confirmation that a creditor or person is in the dataset concerning the log data. Therefore, for this combination of reasons, the risk is deemed negligible.

5.2.2 Preserving characters in strings

The algorithm preserves the length plus all white space and non-alphanumeric characters when anonymizing strings. In this analysis, white space characters are considered non-alphanumeric characters as well. When an attacker has knowledge that some creditor could appear in log data, he could deduce that a certain creditor appears in log data by a unique combination of preserved non-alphanumeric characters. For example, lets take the column $CreditorName$ which in some row contains the string "Het NIC B.V.", which has been anonymized to "oB7 Hds I.c.". The positions of non-alphanumeric characters in that string is the set of $((" ", 3), (" ", 7), (".", 9), (".", 11))$, where each element represent the character with its position. If that set is unique for the column $CreditorName$, i.e. no other string has the same set of indexes of non-alphanumeric characters, and the attacker also has that exact name in his dataset, he can make a match and conclude that the original value must have been "Het NIC B.V.".

To analyze the risk of this type of attack, the representations as shown in the NULL and boolean disclosure risk assessment can be reused. It is possible to find the amount of unique sets of non-alphanumeric characters for any column combination in the representations as follows:

- Query the invoices table and perform a distinct select of all columns in the column combination.
- For each row in the query result, find and save the string length plus the character and position of each non-alphanumeric character.
- Iterate over these results of each row and count how many identical occurrences it has. If the count is 1, then a creditor can be uniquely identified by its length and set of characters and positions.

A preliminary test on dataset A revealed that only the $CreditorName$ column on itself already contains 124 creditors that can be uniquely identified in 8000 invoices. This means that analyzing more columns will result in at least the same amount but most likely more uniquely identifiable creditors. Therefore, the chance of this attack succeeding is higher than with NULL and boolean disclosure.

However, the privacy risk is negligible because of the same reasons as listed for NULL and boolean disclosure. Even if the attacker has a confident match, all the attacker learns is the confirmation that a creditor or person is in the dataset concerning the log data.

5.2.3 Dynamic anonymization attacks

As defined in Section 5.1, the minimal range a developer can use for a number is a range of 10, for a 1,2 or 3 digit number. Giving developers such control over anonymization does introduce some possibilities to misuse the ranges to discover original values.

Assume that a developer is the attacker and has access to the anonymized log data and is able to query them with ranges. From the attackers perspective, in the best case scenario, the original value can be discovered in 2 queries. For example, let the original value be 5. Then the attacker executes a query with a range of $[-5, 5)$. The anonymization tool will then show a value in the range $[6, 16)$. If the attacker then queries with a range of $[6, 16)$ the anonymization tool will show a value in the range $[-5, 5)$. Now the attacker knows that the original value is 5, since there is only one number that will have these outputs for those ranges.

Discovering the original value in 2 queries can only occur if the attacker is lucky or has some knowledge about what the original value could be to guess more accurately. A more realistic approach to an attack is the use of a sliding window of ranges. The anonymization tool will show if the original value of a number was in the ranges $[0, 10)$, $[10, 100)$, $[100, 1000)$, etc. In the case of an original value in the range $[0, 10)$, an attack could be performed by querying all ranges from $[-9, 1)$ to $[9, 19)$ and saving the result each time to discover the original value.

Let r be the minimum range of the anonymization algorithm. Regardless of the minimum range the attacker can use, he needs at most $r + 9$ queries to discover the original value. Increasing r only linearly increases the number of queries needed. An attacker could construct a script that discovers original values with relatively low effort, especially considering that the attackers are developers. Increasing the minimum range r is not the way to solve this problem, because then the utility of the log data would be significantly affected. An organizational method will be used instead: developers can only access each log data x times. The advice is to keep x as low as possible and for now we advise $x = 3$, such that developers can look at a log data normally, then enter ranges and adjust them once if they need to. Using a value $x \geq r + 9$ will reintroduce the risk.

The described attack scenario could also be used for dates (with time), since developers can use dynamic anonymization for dates as well. However, since the minimum range for dates is 10 days, it will cost more queries to discover the original hour, minute or second. Therefore, we can use the same organizational methods as described for numbers to mitigate this attack scenario.

5.2.4 $type in Json.NET

The anonymization component uses Json.NET [7] for parsing and altering JSON. To correctly deserialize the JSON, the type of information must be saved for the ViewData and TempData dictionaries. Both dictionaries have values of type "object", C#'s most high level object type, meaning that anything can go in there. Json.NET would not know into what concrete type the values in these dictionaries need to be deserialized in, so therefore the type information is saved. Json.NET does this deserialization by saving type information in a JSON field with "$type" as key. An example is shown in Listing 4.

Listing 4: Example of $type in Json

```
"$type": "System.Collections.Generic.List`1[[Application.Message.
    UserMessage, Models]], mscorlib",
"$values": [
{
  "UserId": 9,
  "Read": false,
  "CreatedDateTime": "2019-02-15T10:23:10",
  "SendBy": "Employee1",
  "Message": "Test message"
  "Id": 1
}]
```

Since $type is needed for deserialization, every field with that name is skipped by the anonymizer, which can be a vulnerability if developers give privacy sensitive fields the name "$type". In C#, variables and properties may not start with a $ so there is no risk there, but in JavaScript they can. However, ViewData and TempData is server side only, so there is no JavaScript involved in them. The only place where $type could be misused is where JavaScript sends an AJAX request to the server with post data where the key of a value is $type. However, the web application library has support for blocking certain fields in POST data, originally intended to not save passwords in the log data. If $type is added to the list of blocked fields, then this risk is no longer a problem.

5.3 Conclusion

We successfully realized every anonymization requirement. Permutation is chosen as the main anonymization technique. Furthermore, we described the high-level anonymization algorithm and thereafter anonymization for every data type in detail. As an example, the first invoice from Listing 1 can be anonymized to the invoice as shown in Listing 5.

Listing 5: Basic anonymized invoice

```
{
  "Id": 3957,
```

```
  "EntityNr": "47",
  "CreditorNr": "836",
  "System": "5WbwC",
  "CreditorName": "xxAwMSChlGU p.R.",
  "InvoiceDescription": "O3eQlZ etC wDDyWj85 EpTd9 tH3OtsO4C vhj
     etwOAWOsI 5CDJ",
  "Address": "n47M8PUSCFcHu8iy tfq",
  "Postalcode": "y0ow NV",
  "City": "rm2nrLr7",
  "IsRisk": true
}
```

The requirements also describe string preservation rules, the disclosure of booleans and NULL values and the dynamic anonymization. Since these are not without privacy risks, we performed risk assessments. Only in rare cases of combinations of boolean and NULL values could a creditor be re-identified, but then only the existence of a creditor is revealed and not any other details. The dynamic anonymization attacks are not solved with a technical solution, but instead addressed by introducing organizational protection in the form of access limitation. The $type risk is addressed successfully.

While the risk that comes with the string preservation rules and disclosing boolean and NULL values is deemed negligible, disclosing them does cause the algorithm to not be able to guarantee full anonymity. The same holds for the dynamic anonymization attack window, how tiny it may be, there is no full anonymization. The current anonymization is sufficient for Het NIC and most customer contracts, but they may introduce some other organizational restrictions such as the need for permission before looking at any log data to ensure that only trusted employees access the log data. Furthermore, all access to the log data is logged, so in case of an attack, the attacker can be identified.

6 Evaluation

The anonymization method produces anonymized log data. Since the key goal of the log data is to aid developers when solving bugs, the next step is to test whether this anonymized log data is still useful for developers. An empirical evaluation will be held for the end users, meaning that the complete system will be tested.

We now have developed a new logging system which is able to automatically log user requests and all data belonging to those requests. The requests themselves can be queried via Kibana. The data belonging to the requests can be queried via the web application and is anonymized before leaving the ElasticSearch server. We developed this new system for the main objective of this thesis: extending the log functionality of APRA, such that more information is captured about data on live environments in order to aid developers when solving difficult bugs that cannot be solved with traditional logging information.

We held an empirical evaluation in order to test whether the developed solution does indeed help the developers. There are two options to evaluate the complete logging solution. Firstly, a field test could be conducted. The logging solution could be deployed and the developers would use the component to aid in solving any new incoming bugs. The second option is to prepare a set of bugs and sit down with the developers while they solve them. The first option offers a more realistic test setting, however, due to time constraints, the second option is chosen instead. To evaluate whether the new logging system is helpful for developers, the evaluation will need to answer the following questions for each prepared bug:

- Were the developers able to find the log events belonging to the bug?
- How did the developers use the log data?
- How did the log data help the developers in the process of solving the bug?

The first question will address the intuitiveness of the system, while the second and third questions addresses the utility of the log data. This section will first describe the method and bugs used for the evaluation and thereafter the results in which the questions will be answered per bug.

6.1 Method

The prepared bugs will be recreations of bugs that have been reported in APRA, the live web application running for Het NIC, by simply creating a branch in the git repository of APRA which is rolled back to the latest commit before the bug occurred. Using real bugs is more realistic than fabricating bugs where it is known that log data is useful.

We choose to include a bug in the evaluation when it is related to data in some way, either data that is generated in the front end or data returned from the back end. Bugs that are not related to data are not included since we can know beforehand that no log data is needed to fix them and therefore the do not add any value to the evaluation. Furthermore, the chosen bugs are as diverse as possible, such that the utility of the log data is tested with much variety. Among the chosen bugs are bugs which are expected to be solvable with the help of the log data, but also bugs for which it is likely that the log data is not sufficient. Lastly, the chosen bugs are relatively recent, the oldest is bug is 3 months old, since the branches which contain older bugs are deleted for APRA. We will now list each bug with an explanation of their cause.

Bug 1, comma

Whenever I add an unknown credit note (statements creditor assessment) and enter 2560,99, then the input gets saved as 256.099,00. So it seems that if I enter a comma in the number, it automatically gets appended with ,00.

The cause of bug 1 is that the back end expects and processes the collectible value as an integer while the back end should expect a double. The first question developers are going to ask themselves is most likely whether the problem is front or back end related and the log data could possibly help in answering that question, since its preserves the type during anonymization. It is also interesting that the view where this bug is located is at the end of the statements process, since it will take developers at least 15 minutes to set up their local environment to debug the bug.

Bug 2, selecting

When making the selection of statement creditors, the currently selected creditors cannot be selected after deselecting them. When a creditor on the "To research" tab is unchecked, it will not be included in the creditors list that is being researched. However, if I check that same creditor again, it is not marked for research.

The front end is at fault with bug 2. It always posts `Research: False` to the server while it should alternate between true and false (when selecting and deselecting the research-checkbox). This behaviour can be seen in the log data because boolean values are disclosed.

Bug 3, collectible value

When adding an unknown creditor note (statements creditor assessment), the collectible value is always 0 after saving.

Bug 3 is located in the same view as Bug 1. The cause of this bug is simple: the collectible value is not present in the view and therefore will be saved as 0 in the database. In the log data can be seen that the field "collectible value" is missing, since that field is never posted to the server.

Bug 4, calculation

If I click on the calculate button in the statement parameters view, the amounts "Selected creditors" and "Unselected creditors" change. However, if I click on the save button then the amount "Unselected creditors" changes again, while it should stay the same. This issue can have 2 causes: either the calculation of the amount of creditors is bugged on the front end side or the saving process is bugged server side.

While the fix for bug 4 is short, replacing a variable name in a calculation in JavaScript, discovering the fix is difficult. The log data will show the actions when the user clicked on the calculate and save buttons, but not with the real values, which a developer most likely needs to see in order to trace down the faulty calculation.

Bug 5, project manager

As project manager I want to go to the OPL page of a project owned by another project manager but then I get an error page.

The cause of bug 5 is a method call with a boolean set to false while it should be set to true. Using the stack trace in the log event the developers should be able to discover the cause fairly easily.

Bug 6, background task

Running the "ORG suggestions" task fails for database 2018092. The following steps reproduces the problem:

- *Run project creditor analysis*
- *Run task "ABS suggestions"*
- *Run task "BTW suggestions"*
- *Run task "ORG suggestions", which then fails*

Bug 6 is challenging. A background task, the "ORG suggestions" task, fails and its exception is saved in the database. A background task is a long running process that runs on a different thread such that the web server is not slowed by the heavy calculations of these tasks. Since background tasks have their own exception handling, the exception will not be present in the log events since only uncaught exceptions appear there. The exception is available in log data, since it is exposed to the user on a view and therefore recorded, but that will be anonymized. Furthermore, developers can not reproduce this bug since database 2018092 is not available for them since that is a real database used in production. The expectation is that the developers can not fix this bug.

Each bug will have its own code branch, such that they can be reproduced and solved individually. By reproducing the bugs on each branch ourselves, there will also be log events in Kibana for each bug, such that a real production setting is mimicked. Each participating developer will be given the six bug descriptions, access to Kibana and access to an instance of APRA with which they can look up anonymized log data belonging to log events.

6.2 Experiment execution

Before the developers start solving the bugs, they were allowed to practice with Kibana for a few minutes. Since the bugs are recreations of real bugs from APRA and the developers that will participate in this test are all developers from Het NIC, it could happen that a developer already knows the solution to one or two bugs. In that case, these bugs will be used as examples to show how Kibana and accessing log data works. Otherwise, Kibana will be explained with log events and log data irrelevant to any of the six bugs.

When the developers understand Kibana's searching mechanisms, they can start solving the bugs with the help of the logs. Their usage of the logs will be recorded and analyzed. A bug counts as solved when the developer knows what part of code needs to be fixed, since thereafter, the developers will not need the log data anymore. To save time, actually solving and testing the solution of the bug is not a part of the test. Each bug will be accompanied with a date and time to give the developers a hint as to where to search for the log events related to the bug. When a developer is finished with the bugs, a short discussion will be held to discover what they think of the logging system, its usability and if they missed any information in the logs.

For each developer, it is important to know how extensive their experience with the system is. A developer with much experience could use the log data in a different way than less experienced developers, since they will know where to look for the buggy code faster.

In total, seven developers performed the test. These developers have varying levels of experience with APRA, ranging from 1 week to 2 full time years. Two developers are full time employees, while the other five are part time, which means they have worked less hours than their full time colleagues. Furthermore, the developers differ in their preference for front end or back end development. Some developers have more affinity for back end development.

6.3 Results

This section will summarize the process of developers for each bug and answer the questions indirectly. The report of each evaluation with each developer can be found in Appendix B.

Table 8: Bug solving results with log data per bug and per developer

Bug #	**Times solved**
1	3 + 2 known
2	3
3	5
4	0 + 1 known
5	7
6	0

Developer #	**# of bugs solved**
1	3 + 1 known
2	3
3	3
4	3
5	2 + 1 known
6	1
7	2 + 1 known

Table 8 shows per bug how many times it was solved and per developer how many bugs were solved. Note that in this table and this section, a bug is counted as solved when the developers utilized log data to fix the bug, if any form of active debugging was used, then it is not counted towards the total. This method of counting is chosen since every bug is ultimately solvable when using local debugging to reproduce the problem.

Bug 1, comma

All developers were able to find the log events related to this bug. However, there was a division between the developers when viewing the log data. Because the format of the posted data was visible in the log data, some developers were able to conclude that the bug was back end related. They were then able to solve the bug rather easily. Other developers did not notice the data format and needed more time to discover the problem. This difference in behaviour is not related to the experience of the developers.

Bug 2, selecting

After the developers found the log data, most of them noted that `false` being posted multiple times is odd. While most programmers directly drew a conclusion from the multiple occurrences of `false`, two more experienced developers were hesitant to do so. They were hesitant due to the fact that they could not be certain if the objects in the different log data objects were related to

each other, since their IDs were anonymized. The developers that did draw a conclusion were able to quickly find the problem, however.

Bug 3, collectible value

All developers were able to find the log events and the larger part of the developers spotted the missing field in the log data. The developers who missed that clue did not know the name of the field in the log data, because of their inexperience with the code related to the bug. The bug was easily solvable when the developer had the knowledge that a field is missing, but otherwise the bug was more difficult.

Bug 4, calculation

Firstly, some developers had trouble understanding bug 4. The description is difficult to understand when one has no knowledge of the related code and views, which made the related logs more difficult to find. Secondly, no developer used the log data in solving this bug, due to the fact that all numbers are anonymized while exact numbers are needed. A few developers came close to fixing the bug without debugging, but all eventually needed to debug the bug themselves.

Bug 5, project manager

For bug 5, the log data was not really needed. Only one developer used log data to double check if there was no weird behaviour just before the error occurred. All developers found the problem using the stack trace in the log event.

Bug 6, background task

Most developers were disappointed by the fact that the error message that is shown to the user was anonymized in the log data, but they also understood why. Their amount of research was different, but all came to the conclusion that the bug was currently not fixable. They also gave some ideas on how they would fix the bug the next time it occurred.

6.4 Discussion

This section discusses the developers' opinion of the logging system and lists their recommendations. Thereafter, the overall evaluation results are discussed.

During the process of solving bugs and afterwards in the interviews, the developers' reaction on the system was positive. They liked the system and the ease with which the log events can be found and the log data is retrievable. For this set of bugs, the developers did not use any dynamic anonymization. No conclusions can be drawn from this however, since only some very specific bugs will need dynamic anonymization and such a bug did not occur in the near past of APRA. There is also a difference in how the log data is used by different levels of experience of the developers. Their general experience is not important, but their experience with the code that is concerning the bug is. Developers that have worked on code around the bug could find the correct log events more quickly and also knew where to look for the bug faster. Anomalies in the log data were also spotted more often by the more experienced developers, which means that the log data is more useful for developers that have experience with code around the bug. It is not a problem that the log data is

more useful for more experienced developers, since generally the developers who get assigned to a bug are the ones who have knowledge of the code in the part of the system where the bug occurred.

The developers had some recommendations for improvements as well:

- 1 developer would like to see the roles assigned to a user next to the user ID. Since most requests in APRA are restricted to certain user roles, role information can be needed to reproduce a bug.
- 1 developer would like to see that the number 0 is disclosed. This was also mentioned in the group discussion held in Section 3.2.
- 3 developers would have liked to see consistency between IDs between multiple log data objects. So when they look at two log data objects from the same URL, they would like to know if these concern the same object.

All three points are valid and likely good improvements. However, point 2 would require a new risk analysis and point 3 costs time to implement. Combining the time cost with the positive results of the evaluation, we decided to not implement these improvements in this thesis, but leave them for future work.

Not unsurprisingly, bug 4 and 6 were not solved by any of the developers, since solving bug 4 requires exact data for solving the faulty calculation and bug 6 requires more information to be inside the background task logged before it can be solved. Bug 6 does show however that even with the extended automatic logging system, manual log statements of developers are still important and not every bug is easier to solve with this new logging system. Bug 4 shows that not every data-related bug becomes easier to solve by vieweing anonymized POST and return data related to a request. Bug 4 and 6 are also the reason why no developer exceeds an amount of four solved bugs. Most developers failed to find one bug, which brings the average total on three solved bugs.

For bug 1, 2, 3 and 5, the new logging system was helpful for the developers, since the bugs can be more easily located with the help of log data. Kibana in combination with the log data can save much time with debugging. The log data helped developers solve some of the bugs without debugging. For example, time was saved for bug 1 and 3, where preparing data to debug the bug can cost up to 30 minutes. This evaluation shows that developers' ability to solve bugs is improved by the new logging system. However, the evaluation does not show what happens if the log data is the only source of information for solving a bug, since no bugs in the recent history of APRA are dependent on data in such a way. Only when such a bug occurs can this be properly tested. We do have bug 6, but that one can only be solved after developers add extra manual log statements.

In general, having any extra data that is related to a bug is helpful for developers. More information helps a developer localize, reproduce and then solve a bug more easily. Therefore the log data helps with understanding a bug and the time it costs to solve it. We anonymize the log data, which can have negative impact on the utility of that information. Bug 4 is a clear example of this and bug 1 is another example that real data is more useful, since the real values of booleans are needed to solve that bug and our anonymization algorithm discloses booleans. So therefore, the log events with anonymized log data do help developers in understanding bugs but certainly not with all different kinds of bugs.

7 Conclusion

In this project, we aim to answer the following question:

How to extend basic logging such that developers are provided with more information about a live environment when solving bugs, while preserving privacy of data in logs?

To answer this question, we first discovered what information developers need from logs when they are locating, reproducing and solving bugs in Section 3.1. A small part of the information came from literature, but the larger part was discovered by conducting interviews with developers themselves. In web applications, developers would like as much information as possible related to a request. Therefore, we decided to not only log a request but also, for every request, all POST data and data returned by the server.

Since this POST and return data is privacy sensitive and the developers working at *Het NIC* are not allowed to see that data directly, the next step was to research how data in logs can be anonymized in Section 3.2. This anonymization should remove personal identifiable information from the log data while keeping the log data as useful as possible. The possible anonymization techniques were researched from literature. To discover what information in logs developers are using when solving bugs, a group discussion was held. This group discussion revealed what developers wished to see for each data type. Some of their wishes were unrealistic in terms of preserving privacy, while other wishes were new and good ideas.

The requirements for both logging and anonymization were listed and numbered, after which the development of a new logging system could start in Section 4. We first chose to use ElasticSearch for log storage and Kibana for log querying. With that choice, we could design the system architecture with the .NET Core proxy and web application library. The .NET Core proxy will make sure that all log data that leaves the ElasticSearch server is anonymized. Furthermore, we introduced and explained the replay functionality.

With the logging infrastructure set up, the next step was to add the anonymization method in Section 5. We first explained that we chose permutation to anonymize all different data types. We showed the high-level anonymization algorithm and thereafter the anonymization details of each data type: numbers, dates and strings with its subtypes. As described in the requirements, string length plus white space and non-alphanumeric characters are preserved. Furthermore, NULL and boolean values are being preserved and numbers and dates were given the dynamic anonymization functionality, which means that developers can query whether a number or date is within a certain range. The string preservation rules, NULL and boolean disclosure and dynamic anonymization are not without privacy risks, however. Therefore, a risk assessment was performed for the string preservation rules and NULL and boolean disclosure. The possible attacks were analyzed for dynamic anonymization. We showed that the risk of releasing privacy sensitive information via the string preservation rules and NULL and boolean disclosure is negligible. The attacks on dynamic anonymization were solved with an organizational method: limit the number of times a developer can query the same piece of log data.

We now have a new logging system in place that logs more data related to requests and is able to anonymize that data when developers want to view it. The last important step in the process was to evaluate whether developers' understanding of bugs is improved with the extended logging. We performed an empirical evaluation where we reintroduced existing bugs and let developers solve them with the help of the extended logging. We chose six diverse bugs and let seven developers

try to solve them. We discovered that the developers like the intuitivity of the system and are positive about the log data as well. Most developers were able to solve three out of six bugs using only the log data. Two bugs could not be solved since the log data was not sufficient. In one case, exact (non-anonymized) data was needed and in the other case more manual logs related to the problem should be added. The evaluation showed that the extended logging helps developers in understanding bugs and the time it costs to save one. However, it also showed that there will always be bugs which need more and non-anonymized data to be solved and in that case, the extended logging is not sufficient.

There are some threats to validity to this research. We did interview a diverse set of developers in order to formulate the requirements, but we interviewed only seven developers. Interviewing more developers will help confirm the requirements or could perhaps lead to some new requirements. Furthermore, the prioritization of requirements could change, but that will not have a significant effect on the implementation of the logging system. Similarly, we only held one group discussion with developers from *Het NIC*. To make a stronger claim or perhaps get new insights, more group discussions should be held with developers with different backgrounds. Lastly, the evaluation was performed with bugs and developers from *Het NIC*. Performing the evaluation multiple times with different sets of bugs and people could lead to new results or insights.

Finally, the answer to the main question is:

We extended the traditional request logging with logging all data related to a request. This is data that is posted by the user as well as data returned by the server. Posted data and data returned by the server is privacy sensitive. Therefore, when developers want to access this log data, it is anonymized. Developers can view the log data in JSON format as well as replay a view as a user saw it. Evaluation did not confirm that the developers were able to solve bugs which could not be solved with the traditional logging. Using the new logging system in production could confirm that, but this test could not be conducted due to time constraints. However, the evaluation did show that developers' understanding of bugs improved when viewing the log data via the new logging system. They were able to better localize, reproduce and fix a bug in less time.

7.1 Future research

This section describes future research ideas that were discovered during the execution of this graduation project.

The contracts *Het NIC* has with customers sometimes include a section which states that all data related to that customer must be deleted from the systems of *Het NIC*. This will include the new logging solution. Currently, if we want to delete logs related to a customer, we need to delete all logs until the date that the customers' data no longer appears in the logs. It would be better if we could execute a delete query for a specific customer. This requires that each log event and data is linked to one or more customers. Performing this link during the creation of a log event is not trivial, since data on views can concern multiple customers and there is not always a direct reference to a customer available in the data. Therefore, linking log events and data to customers could be an interesting research topic.

Around March 2019, after the new logging system was realized, Amazon released Open Distro for ElasticSearch [9]. This is an initiative to bring essential features back to the open source community that have recently been made paid features by the developers of ElasticSearch. In this thesis, one of those functionalities is security. The reason why we chose to use SearchGuard is to realize HTTPS

and access control for free for Kibana and ElasticSearch. In turn, we needed to introduce a .NET Core proxy to handle the anonymization since SearchGuard was installed and only one plugin at a time on ElasticSearch can alter requests. Installing Open Distro for ElasticSearch with its open source security plugin would eliminate the need for SearchGuard and therefore an anonymization plugin for ElasticSearch could be realized. Having the anonymization directly on ElasticSearch as a plugin would increase the certainty that no privacy sensitive data leaves ElasticSearch and decrease the configuration management effort since the .NET Core proxy does not need to be installed and maintained. So it would be useful to research how to use Open Distro for ElasticSearch and implement the anonymizer as an ElasticSearch plugin.

The log events plus data provide insights in what data every user sees. This data opens the doors for extensive process mining researches. In the introduction we mentioned that the Dutch Data Protection Authority will periodically check companies whether there are any privacy breaches. An answer to the following question could directly show them that this is (not) the case: how to detect when a user has seen data that he is not supposed to, i.e. a privacy breach? This interesting question can perhaps be answered with the data that is now being logged.

Lastly, two developers mentioned that the best way to debug would be to have a virtual machine dump at the time a bug occurred. Whenever an uncaught exception occurs, the virtual machine state should be saved such that developers can load that virtual machine in their own environments and see the exact state at the time the exception occurred. Therefore, saving the virtual machine state and then spinning it back up at the exact same line of code could be an interesting research topic to realize an excellent way to debug. Naturally, this virtual machine state will contain privacy sensitive information from production environments and this information should be protected in a similar way as this thesis shows.

References

[1] Algolia. https://www.algolia.com/. Accessed: 2019-01-08.

[2] Apache lucene. http://lucene.apache.org/. Accessed: 2019-01-08.

[3] Application insights. https://docs.microsoft.com/nl-nl/azure/application-insights/app-insights-overview. Accessed: 2019-01-25.

[4] Autoriteit persoonsgegevens. https://www.autoriteitpersoonsgegevens.nl/l. Accessed: 2019-03-13.

[5] Elasticsearch and kibana. https://www.elastic.co/. Accessed: 2018-12-20.

[6] General data protection regulation (gdpr). https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1528874672298&uri=CELEX%3A32016R0679. Accessed: 2019-01-07.

[7] Json.net. https://www.newtonsoft.com/json. Accessed: 2019-03-20.

[8] Nlog. https://nlog-project.org/. Accessed: 2019-04-23.

[9] Open distro for elastic search. https://opendistro.github.io/for-elasticsearch/. Accessed: 2019-04-08.

[10] Search guard. https://search-guard.com/. Accessed: 2018-12-20.

[11] Splunk. https://www.splunk.com/. Accessed: 2019-01-08.

[12] Uniform resource identifier (uri) schemes. https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml. Accessed: 2019-01-29.

[13] Wet op de ondernemingsraden, artikel 27 lid 1 i. https://wetten.overheid.nl/BWBR0002747/2019-01-01. Accessed: 2019-01-07.

[14] Leif Azzopardi, Myles Doolan, and Richard Glassey. Alf: A client side logger and server for capturing user interactions in web applications. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 1003–1003, New York, NY, USA, 2012. ACM.

[15] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 17–30, New York, NY, USA, 2007. ACM.

[16] Henry Allen Feild, James Allan, and Joshua Glatt. Crowdlogging: Distributed, private, and anonymous search logging. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 375–384, New York, NY, USA, 2011. ACM.

[17] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[18] Adam Fourney, Richard Mann, and Michael Terry. Characterizing the usability of interactive applications through query log analysis. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1817–1826, New York, NY, USA, 2011. ACM.

[19] Q. Fu, J. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158, Dec 2009.

[20] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 24–33, New York, NY, USA, 2014. ACM.

[21] Benjamin C.M. Fung, Ke Wang, Ada Wai-Chee Fu, and Philip S. Yu. *Introduction to Privacy-Preserving Data Publishing: Concepts and Techniques*. Chapman & Hall/CRC, 1st edition, 2010.

[22] Jaime Teevan G. Craig Murray. Query log analysis: Social and technological challenges. *ACM SIGIR Forum*, 41(2):112–120, December 2007.

[23] P. Goswami and S. Madan. Privacy preserving data publishing and data anonymization approaches: A review. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 139–142, May 2017.

[24] Katja Hofmann, Bouke Huurnink, Marc Bron, and Maarten de Rijke. Comparing click-through data to purchase decisions for retrieval evaluation. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 761–762, New York, NY, USA, 2010. ACM.

[25] Kinshuman Kinshumann, Kirk Glerum, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. *Commun. ACM*, 54(7):111–116, July 2011.

[26] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 106–115, April 2007.

[27] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association.

[28] Di Ma and Gene Tsudik. A new approach to secure logging. *Trans. Storage*, 5(1):2:1–2:21, March 2009.

[29] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. L-diversity: privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 24–24, April 2006.

[30] A. Madani, S. Rezayi, and H. Gharaee. Log management comprehensive architecture in security operation center (soc). In *2011 International Conference on Computational Aspects of Social Networks (CASoN)*, pages 284–289, Oct 2011.

[31] Andriy Miranskyy, Abdelwahab Hamou-Lhadj, Enzo Cialini, and Alf Larsson. Operational-log analysis for big data systems: Challenges and solutions. *IEEE Softw.*, 33(2):52–59, March 2016.

[32] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[33] Mehmet Ercan Nergiz, Maurizio Atzori, and Chris Clifton. Hiding the presence of individuals from shared databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 665–676, New York, NY, USA, 2007. ACM.

[34] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Queue*, 9(12):30:30–30:40, December 2011.

[35] Diego Perez-Palacin, Radu Calinescu, and José Merseguer. Log2cloud: Log-based prediction of cost-performance trade-offs for cloud deployments. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 397–404, New York, NY, USA, 2013. ACM.

[36] William Pourmajidi and Andriy V. Miranskyy. Logchain: Blockchain-assisted log storage. *CoRR*, abs/1805.08868, 2018.

[37] Ariel Rabkin and Randy Katz. Chukwa: A system for reliable large-scale log collection. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[38] Azarias Reda, Yubin Park, Mitul Tiwari, Christian Posse, and Sam Shah. Metaphor: A system for related search recommendations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 664–673, New York, NY, USA, 2012. ACM.

[39] P. Samarati. Protecting respondents identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, Nov 2001.

[40] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[41] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, P. W. Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper , a large-scale distributed systems tracing infrastructure. 2010.

[42] Latanya Sweeney. Simple demographics often identify people uniquely. *Data Privacy Working Paper 3*, (3), 2000.

[43] Latanya Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002.

[44] Roman Słowiński. *Intelligent Decision Support: Handbook of Applications and Advances of the Rough Sets Theory*. 01 1992.

[45] Welderufael B. Tesfay, Peter Hofmann, Toru Nakamura, Shinsaku Kiyomoto, and Jetzabel Serna. Privacyguide: Towards an implementation of the eu gdpr on internet privacy policy evaluation. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, IWSPA '18, pages 15–21, New York, NY, USA, 2018. ACM.

[46] Ke Wang and Benjamin C. M. Fung. Anonymizing sequential releases. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 414–423, New York, NY, USA, 2006. ACM.

[47] Xiaokui Xiao and Yufei Tao. Personalized privacy preservation. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 229–240, New York, NY, USA, 2006. ACM.

Appendix

A Developer interviews

This section includes reports on all the interview held with the developers to discover which (privacy sensitive) information developers need from logs when localizing and reproducing bugs. The reports are summaries of the interviews and not transcripts. We are not interested in the exact things the developers say but more their opinion in general. Where deemed relevant or important, the developers are quoted instead of summarized.

A.1 Developer interview with N. 27-9-2018

I started by introducing myself and explaining to N. how this graduation project originated. N. then told something about himself. He is currently doing his master computer science on TU/e and before that did HBO software development. Next to his master he works part time for *Het NIC* as a developer and architect.

Since he is my colleague, and *Het NIC* does not have an application in production yet, his experience with bugs in live environments is limited. However, there have been some bugs on test environments which the end users use for testing. These bugs could not be reproduced locally or on other test environments.

Bug 1

There was an issue with running a background task. When the task was running, it seemed like the background task stopped logging for multiple hours and then the logs would start again. This must mean that the background task was stopped and restarted. N. said: "We discovered what it caused since we had system logs next to our normal user activity logs. Every time the logs started again after some hours of no logs, some system caches were rebuild which should not have been disposed in the first place. So in this case, the user activity logs were helping us because we could correlate specific URLs (and therefore controller functions) to the points where the background tasks stopped logging". In the end, it problem was that IIS does an application pool recycle whenever it wants, even if there is code running. Only the timestamp and URLs in the user activity logs were used for solving this bug.

Bug 2

There was a problem where sometimes double GET and POST requests would be executed. The bug started with a tester filling in a form to create an object and getting the message that a same object already existed, but the person was absolutely sure it was unique. After investigating the logs, it became clear that the form was posted twice at exactly the same time, where the second request returned an error code since it was not unique. In the end, the bug was solved by ruling out each component of the system until the network was left and then they discovered a wrongly configured firewall was the problem. In this case, the timestamp, URL and HTTP method were important in the user activity logs.

Ideal situation

Then I asked him about the ideal situation. N. started with: "Of course, the ideal situation differs per bug, but I can imagine that some information can always be helpful. If I start investigating a bug, I am going to try and trace back to the root cause. I would like to have an idea of what was on the users' screen and how he acted. This will also give an idea if something was wrong some steps before the bug actually occurred. It would be nice to have data in these steps as well, so you can trace back to the root cause." Next to that, he would like some kind of summary on how the system was performing at the time the bug occurred. Since a bug can also be caused by the server or network, for example (as is the case in the bugs described above). Furthermore, he would like to know if around the same time other exception were thrown as well. In other words, is the bug localized or are there more failures? I followed up by asking: "What if, for example, clicks of the user are being logged?". N. responded with: "Of course, the more information the better. A screenshot could also be a thing. There can be front end errors. Basically you want to see the data that has been exchanged."

To finalize, I asked whether N. had anything to add and he concluded: "Because of the current state of the product, I am helped more by system logs. However, if we have a stable application running, then it could be the case that user activity logs are more important".

A.2 Developer interview with S. 1-10-2018

To start off, I introduced myself and explained the setting of my graduation project. I asked S. to briefly tell something about his experience as a software developer and he informed me that he has 4 years of professional experience as a full stack .NET developer. S. is an indirect colleague, since he is a consultant who works remotely for *Het NIC*. He has done one project before working for Het NIC, which is currently running and being used.

Bug 1

S. explained there was a bug in a web application when a page tried to retrieve data form a web server. Firstly, the logs were used to check which URL was responsible for throwing an error. S. said: "I started looking at the log to find out which method in which controller was responsible for throwing the error. Then I dug further and tried to find the specific parameters with which that function was called". I asked whether these parameters were in the logs, but he said that was not the case. However, he would have liked that, since it would have saved him time in finding out which set of parameters were responsible for the bug.

I continued asking if S. had anymore examples and he told me about the background task bug N. also explained. S. could not remember anymore bugs on production where he had to use logs.

Ideal situation

I gave the example and asked S. to think of his ideal logging situation. As a start, S. said: "the parameters as we talked about, the exceptions including their stack traces. Perhaps also some information about the user, his ID and roles and rights." I did a suggestion to log the JavaScript errors and S. agreed with me that this could be useful. Then S. continued: "maybe the server load at the moment of the bug. System information. Some bugs need this information. To fully reproduce a bug, you would need to save the whole form in your log". I assume he was talking

about a POST request in this case. I tried to get more information by asking what specific data he is interested in next to the POST data. S. replied: "the data so you can see what the client send and what the client received". As a follow up, I asked whether he would like to see some user flow and I gave an example of being able to see the 10 steps leading up to the bug. S. thought this can be added value when solving some types of bugs.

As a last question, I asked whether he had anything to add. S. thinks it is important that there is some kind of GUI to query the logs. Especially when the volume of logs is huge. Specifically, he was thinking about querying on user and time frame. When he has to look at the lines of logs, he would maybe like some colour scheme to be able to quickly distinguish the types of logs from each other. S. said: "this is perhaps equally important as the logging itself".

A.3 Developer interview with D. 02-10-2018

D. has 3 years experience in ASP.NET, MVC, Xaml and Xamarin. He has experience with a framework to obtain logs from devices running a Xamarin application.

Bugs

D. told me that he could not think of a specific bug. Instead, he explained how he usually handles bugs in production. Sometimes, it is difficult for him to localize a bug with the information he has from the logs. D. said: "usually I have to sort of guess what the user has been doing just before the error occurred". So he is not able to easily reproduce the bug. D. explained that they work with invoices and in some logs the ID of an invoice is logged. With that he is able to look up the audit trail of that specific invoice, which he is authorized to do. This means he looks at privacy sensitive information to solve a bug.

I asked him which information he would like to see in these logs in order to make it easier to localize and reproduce these types of bugs. D. responded with: "perhaps it is useful to see the history of the related invoice in more detail. What were the values in the invoice before and up until the bug". In other cases, he said that there was too much information in the logs. What D. means by this is the fact that there are a huge amount of logs in which he has to find the logs relevant for his bug. So he would like to be able to run some query over the logs to select relevant logs.

Ideal situation

Next to the things mentioned above, D. explained that he has used a JavaScript plug-in to record how users click on advertisements. He thinks this could be useful for bug reproducing as well, since you can see if the user has had any odd behaviour when the bug occurred.

He also thinks it is useful to log the input and output of the server, in other words, what is posted by the user and sent to the user.

I asked him: "what if a customer calls with a complaint that the web application is responding very slow"? D. thought it would be a good idea to have some way to detect anomalies. For example, an abnormal amount of request within a certain time frame. It would be nice to detect and report this automatically.

A.4 Developer interview with Sj. 02-10-2018

After the introduction, Sj. told me that he graduated from HBO 3 years ago and has since been working with .NET techniques. In that time, he has done 3 projects. One with health care institutions, which was in production in some municipalities, another one with the largest VOIP provider in the Netherlands and the last was developing a proof of concept for smart energy meters.

Bug 1

During his first project, they had an application that processed so called messages from health care institutions. This means the system handles medical information, which is privacy sensitive. Sometimes, errors would be thrown because of a bug during the processing or bad input from the health care institution. In theory, these errors can be reproduced in a developer environment, but you do not want to have that privacy sensitive data on your laptop.

The first step was to fix the logging, since there was no logging for some locations or the logs did not contain enough information. To be able to find the cause of an error, he added the context to the logging. What Sj. means by this is to log some kind of identifier in order to be able to correlate to log to a message.

Then when an error occurred, Sj. would look at the logs, see the identifier and would look at the object in the production database. I asked whether he was allowed to do that, since it involves privacy sensitive information. Sj. confirmed that he was allowed to do that, since the contract allowed the developers to look into the data when needed.

In the end, the fix for these errors was often to better validate the input from the health care institution and give them clear responses about what was wrong with the input.

Ideal situation

Sj. explained that he recently worked on a proof of concept for smart energy meters. The idea was to let mobile phones read information from these meters, up to 256 simultaneously. To be able to make sense of a log from that system, he again needed some kind of context. So he logged the mobile phone number and also the ID of the smart energy meter.

Furthermore, Sj. likes to use a tool which highlights your individual logs in order to easily recognize the logs you are looking for. This saves large amounts of time.

He would also like to have some kind of system monitoring, like CPU usage and request durations, to be able to see if there were any anomalies during the occurrence of a bug.

Sj. mentioned an ELK stack, which are tools to give insights about your logging. It is nice to have a dashboard which gives summaries and analytics with one glance.

He also thinks it is important to have a framework for logging such that developers can easily log without too much effort.

I then asked whether it would be nice to log the data a user has seen in a web application. Sj. responded: “in these situations, there is a large chance you are logging privacy sensitive data”. I told him that he did not have to worry about that in this hypothetical ideal situation. Then Sj. said that it would be very interesting to be able to see that. He would also combine this with audit logging. Sj. said: “maybe it is possible to see which data has been leaked in the case of a data leak”.

As a last point, he mentioned that it would be nice to be able to combine logs from different micro services. Each service logs autonomous and often includes some kind of ID in the logs. But

as data goes through these service, which are asynchronous, time is often not enough to know which logs of different services belong to one process. If you are able to combine these logs, it is possible to see the path a data record has taken through different services.

A.5 Developer interview with Da. 10-10-2018

After my introduction, I asked Da. to tell about his experience as a developer. He is 33 years old and has been programming since he is 12. Eventually he studied HBO developer and started programming in Java, which he is doing for 8 years professionally now. Currently, he is working on a web application for high schools which handles each request independently, so called stateless.

Da. explains that when he works with logging, he is interested in two high level types. The first type are the system metrics. How many users are logged in, request durations and CPU usage. The second type is the application logging. The first thing he looks for in application logs are the stack traces. Java has extensive stack traces and you can easily click those stack traces to go to the line of code which produced the error. These stack traces are produced by both unforeseen errors like null pointers, and foreseen errors like invalid input. Da. thinks that it is possible to solve 99 out of 100 bugs with the stack trace alone.

Da. tries to solve the bugs test driven. This means he first writes a unit test that causes the bug and fails, then he fixes the code and the unit test succeeds. Having that test in place makes it so the bug will not occur again.

Furthermore, he thinks it is hard for developers to write useful logs. He often encountered logs in the sense of: "Something went wrong here". Those logs have no use at all and only cause noise in the collection of logs.

Da. also mentions that he thinks logging is a sensitive subject between developers. "If you ask 100 developers how and what you should log, you will probably get 100 different answers, just like with how and what to test".

Bugs

Da. explains that they work with two main development teams. The first team develops the program, the content player and the second team develops the content. Sometimes content delivered by the content team contains unexpected information and that causes errors. Recently there was content where some metadata information was missing, which caused a null pointer. To fix this, they needed the stack trace in the log. In another case, the software did certain assumptions about input which it should not have done. This resulted in 300.000 lines of logs in 15 minutes. Eventually, they discovered these logs were many of the same and the error could be resolved fast.

Da. furthermore explained their front end logging: "we do log on the front end when there is a JavaScript error due to wrong user input. Even though we use a front end framework, we had to build something ourselves to send these errors to our log server. On this log server we can query for front end errors ". I asked him whether they sometimes replay bugs. Da. responded: "when we get a call from the help desk about an error, they often now the steps to reproduce it. Then we see how the user clicked through the application and can reason what went wrong. I think the most important thing for a developer is replaying the bug in the code. If that succeeds than fixing the bug is often peanuts". To do the replay, they often have some context information, but they do not necessarily need to know who caused the bug. That is a way they prevent logging privacy sensitive information. Da. admits that in some cases he would like to log more personal information to reproduce bugs if that was legally allowed.

Ideal situation

I asked for Da.'s ideal logging situation: "we work with a system where there is a thread for every HTTP request. I think in the ideal situation you completely log the state/memory of a thread when an error occurs. Then you could exactly reproduce the state at the moment of the bug in your development environment.

Furthermore, DA. thinks it would be helpful to have all information of the request header. In the case of a database error, you want to see the query, but also the result of the query. In other words, the context information.

I asked him whether you would need to log the state/memory of a thread at every step to reproduce a situation perfectly. Da. responded: "I think there are two key factors in that case. The first one is the stack trace, through which methods the code went before coming to the point of the crash. You can combine that with the memory dump to pin point what is exactly the problem".

He further added: "what could be of interest for your research is the fact that different types of applications have different logging wishes. If you have a monolith application, combining the stack trace with a memory dump could work nicely. However, if you have some micro services that combination is perhaps not enough".

As a last thing he added: "I think it could help you to define what is logging and what are metrics. If you want to know why something crashed, then you would look in the logs. If you want to predict if something will break soon, you want to keep an eye on the CPU usage and memory usage".

A.6 Developer interview with J. 10-10-2018

I introduced myself and J. introduced himself. He studied for HBO developer, where he quit 1,5 years later to study computer science on the Vrije Universiteit. He finished his bachelors and started working with Java and has always done so. In total he has around 13 years of professional experience. Currently, he is working on a web application for high schools which handles each request independently, so called stateless.

I explained to J. that I also interviewed attorneys about the GDPR next to developers. He reacted with: "I came across this some time ago. I wanted to update the logging to also include the session ID. However in our case, the session ID is privacy sensitive information and I was not allowed to log it. So to fulfill both wishes, we ended up logging the last 6 characters of the session ID. This was enough to be able to follow the session ID through the logs but not enough to identify a person with".

The bugs

J. explained that they have a central system for the entire organization to log in. This system has a staging and production environment, but because of technical reasons only a limited test environment. This means that sometimes bugs are discovered too late, in the production environment. I asked him which kind of logging this system has. J. responded: "we have an access log, which means that a large part of each request header is logged, plus we have some application level logging, mostly exceptions. It would be nice to have good flow logging as well, to better reproduce what a user has done but we do not have that currently. These logs are send to a central logging server for the entire organization".

He then elaborated on how this central logging server works: “this server handles around 100GB of logs each day. These are almost solely warning and error logs. We have peaks of 6000 users logged in at the same which results in around 200 requests per second”. This is only for his department and he estimates that the peak is around 10.000 on any given day. They have a retention time of 2 weeks, which is already on the short side in his opinion. They do have an option for production environments to enable trace logging, which logs in greater detail. They only do this when they are unable to reproduce a bug locally in order to get more information.

J. explains that they move more and more logic to the front end, because they have a single page application, and are currently figuring out on what to log there and in which detail. In general, they solve bugs in production with the help of logs and system metrics.

I asked J. whether they have any data records or links to data records in these trace logs. He explains that sometimes they add that information, but you are not allowed to log any personal information. It is up to the developers to keep this in mind when programming a log statement. J. admits that this situation is not ideal.

I also asked how they are able to find relevant bugs in those enormous amounts of logs. J. told me that they use an ELK stack with Logstash (https://www.elastic.co/elk-stack) which gives them the ability to query these logs fast. They can query by tag, level and more.

Bug 1

There was a bug where content creators had released a piece of content which should not be released. The title of the content even contained “DO NOT PUBLISH”. J. said: “users are of course curious, so they click on that title when they see something like that and that resulted in errors on production”. They had logged the ID of the content, which is how they discovered that there was something published which should not have been. J. confirms they are allowed to see data in the production database, while they have some privacy sensitive information of high school students in their system.

Bug 2

J. explains that this bug was discovered by looking into the logs. The logs showed that the same HTTP connection was closed two times, which gives errors. With the help of the stack trace they found that their error handling was at fault. When there was an error, the error handling closed the connection and at a later point in time the connection was closed again by default.

Bug 3

There was a user which was not able to log in anymore. Looking into the logs J. saw that the user somehow created two sessions, because there was a duplicate key exception. When they checked the database, they saw that there were two sessions for the user but the code only expects one session. Ultimately, they decided to not fix this bug since it would require much effort while being very rare.

J. elaborated more in general: “I think logging is extremely important for us. When a user calls for help with a bug, you often take a look into the logs to see which path they have taken to cause the bug. I think what is really valuable is the ability to see which path the user has taken through the application. In the end, one request fails, but that could be due to some wrong input or response in an earlier request”. I asked whether he thinks there is enough information in these

logs. J. responded that in most cases, there is enough information, but if they need the context, they add that specifically. The downside of this reactive method is the fact that there needs to be a new release and then the bug needs to reoccur.

Ideal situation

J. would like an audit trail including the select queries in a database. Then some steps are performed on that data before showing it to the user and each step could be logged. J. said: "even better would be to log the complete Java Virtual Machine (JVM) of the server at the moment of an error and then you could just load that JVM on your development environment. Save the state at every step into your code to be able to load a specific step for reproduction".

I asked whether he would like any system information, like CPU or memory usage. J. responded: "that would certainly be helpful, we already have that in some form. Maybe you could do a complete memory dump next to saving the state of the JVM for even better reproduction".

I summed it up by asking: "playing back what the user has done seems to be the key here"? J. confirmed that this is indeed the case. He would like to be able to see a few steps before something went wrong. I asked further: "but that is not only back end logging but front end as well, right"? J.: "yes you are correct. Perhaps you should save the state of the browser and include a memory dump and send that to your log server. Maybe you need to see the plugin list of the browser, because that can also cause unexpected behaviour".

A.7 Developer interview with M. 10-10-2018

M. has a different background than the usual developer. He comes from a more physical side, having developed things as eye scanners before becoming a software developer. He currently works on a web application with front end AngularJS and back end Java. I interviewed him via email and he responded with comprehensive answers, having different bug examples from different companies.

Bug 1

There was data corruption because of an incorrect identifier. What made it hard to solve this bug was the fact that there were no logs of the system in which the bug occurred. Luckily, the bug was also visible in another system in the chain, which used an XML export of the system M. worked on. This system logged a stack trace, which was given to M. This stack trace showed that an identifier had the value "undefined". To reproduce this bug, they set up a test environment and that showed that the bug only occurred on newly made objects. Eventually, they compared the responses between the back and front end of the test and live environments. This revealed that the back end returned a view of an object without an identifier. The front end needed this identifier but instead of throwing an exception, the value of the identifier was automatically set to "undefined". This identifier was then posted to the back end which caused the bug. When trying to find out where the value "undefined" occurred, they needed to check the database. They compared old and new values and with that information concluded that the bug only occurred with newly made objects.

Bug 2

This bug caused a system crash due to not being able to process certain input data. This time, the system was configured to log many things. Logging and audit trail functionality were required by law for this application. The source, time and channel were logged by incoming data and with every step in the process the location of the data being processed was logged.

To solve this bug, the logged time was the first give away. The bug occurred once every week at almost the same time. M. concluded that the bug must have been caused by a specific event. Since they logged each step in the process, they were able to see until which step the processing worked. Eventually, they found out they used certain data to query data from another system while that system did not have that data. Logging the location of the data that was being processed helped in discovering which data was wrong. Without the logging, M. would not have been able to solve this bug. The original data was stored only temporarily in the database (24 hours), before being deleted because of privacy sensitivity. So they had only limited time to search in the data to the cause of the bug.

Bug 3

This is more of an example on how not to log. In this case there was a very unstable, multiserver and multithreaded system which crashed frequently. M. joined this project in a later stage. He explained that the developers before him logged way too much information (log statements were added everywhere because the system was so unstable) and did not categorize them (trace, info, warning, error). There was no DTAP environment which meant development and debugging happened on the production environment. This basically meant that the only way to debug the application was to add even more logging. The system was responsible for handling several hundreds of thousands of requests each day and due to the over extensive logging, sometimes the systems would run out of disk space multiple times per day.

M. explained that localizing a large part of the bugs was simply not possible. The system existed of queues which used up to 20 threads per queue and having around 14 instances running, it was not possible to know which server, queue or thread caused the bug. Next to that, it was not possible to see the flow of a request through the application because the logs were spread over multiple files. In the end, they decided to rewrite the whole application on a different architecture.

So, the logs did not help in solving bugs. M. explained that the logs were mainly used to check if the system was running. It became so bad that the company hired employees to watch the log and restart the system if they saw a certain kind of log statement often.

Ideal situation

I sketched the hypothetical case and M. answered extensively:

References to relevant objects in the database: M. would like to see references to objects in the database that are relevant at the moment of the log. That way, you are able to view these objects if you need them during reproducing the bug.

URL including parameters, Request bodies and HTTP response codes: M. would like to see the above mentioned things in the case of a web request to the server. He says: “to prevent too much overhead, I would log 4xx and 5xx responses from the web server in any case”. This is because the most bugs he fixes have to do with interaction between front and back end.

If he has the information of the HTTP request, then he can determine more quickly whether the error occurred due to wrong data being send to the system or if there is an internal error.

User ID: In M.'s experience, some bugs only occur for some users. If there is a user id in the logs, then you can better follow what that user has done when reproducing the bug.

Screenshots of the front end: The front end framework M. currently uses saves a screenshot of the browser the moment a test fails. This allows him to follow the path the test has taken. If this could be done for production, that would be nice. It also helps with vague bug reports from users.

Everything in the request header

OS, browser version, etc.: Some front end bugs rely on this information.

Class, method and line number: This kind of works like a clickable stack trace. It will give you the ability to go to line of code for every log and not only with exceptions.

Application version: M. explains that he sometimes realizes that a reported bug has already been fixed in a newer version. Having the application version in the log will tell the developer in which version the bug occurred and then he can check whether it has been fixed already.

Server/node: In the case the organization has its own dedicated servers, it could be useful to log which server/node/machine produced the log as some bugs are hardware related. If you have an environment where every instance of your application is exactly the same, then this becomes less relevant.

B Utility tests

B.1 Bart

Bart is an intern and third year Fontys student who worked part time on APRA for about 10 months, giving him around half a year of full time experience.

- **Bug 1**: Coincidentally, Bart already solved this bug 2 days prior to the test. So he knew exactly where the problem was. He did have a look at the log data related to this problem and said it would probably have saved him some time, since he could directly rule out that the front end is at fault here with the help of the log data.
- **Bug 2**: Viewing the log data, Bart was able to quickly see that the problem was front end related. He went to look at the view and corresponding JavaScript and was able to solve the bug without running his development environment.
- **Bug 3**: Bart noticed that the collectible amount was not present in the log data, so he went to take a look at the view. He said: "the log data gave a really good clue on the location of the problem. I did not need to debug to discover that the field was missing from the view".
- **Bug 4**: Bart looked at the log data of three different POST events and his first impression was that the numbers were indeed different. However, it took him a few moments to realize that the numbers were randomized and he had no way to know if the values were actually equal or different. Therefore, he booted up his development environment and started to fix the bug by debugging himself.
- **Bug 5**: Having found the exception data in the logs, Bart found the problematic code in around 5 minutes without having to reproduce the bug locally.
- **Bug 6**: Bart was able to find the log data with the exception, however, the message was anonymized. He was able to deduce that the problem was located in the execution of the task and not in the start up of the task. He tried reproducing the bug locally by using development databases but the task did not fail for those databases. Bart concluded that the fault must be related to some values in the 2018092 dataset, but since he does not have access to that dataset, he was not able to fix this bug.

During the discussion, Bart mentioned that he did not expect that the log data would be so useful. Part of that reason could be that these bugs were specifically selected to be related to data. He said: "I was actually more helped by the log data than by the log events. For these bugs, I just used the log events to search for the log data". On the other side, he did not see where or when he would use the constraints he could apply to the log data. As a last remark, he would like to see the user roles in a log event, because he has no way of knowing who is behind which user ID.

B.2 Edwin

Edwin is a new colleague. He just started working on the system after graduating from Fontys at *Het NIC* for a different project. This means he has about 1 week of experience. In general, he needed more time to find the log events belonging to the bugs.

- **Bug 1**: Even though the bug description does not mention which field gets the wrong value after adding a creditor note, Edwin was able to conclude the problem must be back end related, since the formats of all numbers seemed right. After some searching in the back end he was able to find the erroneous data type.
- **Bug 2**: Partly because of his inexperience with the system, it took him some time to find the correct log events. After examining multiple log data objects, he thought that the problem should be front end since `false` is posted every time. He navigated to the view and quickly spotted the incorrect line of code.
- **Bug 3**: Edwin did not know which field he should look for, so he did not notice that the log data was missing a field. After some searching however, he did discover the missing field, since all other fields seemed to be posted correctly. So he knew the view must be the place where the bug is located. In the end, he was not able the solve this bug with just the log data and had to reproduce the bug locally to discover that the form is missing the field.
- **Bug 4**: For a new person, the description of this bug is vague. Combine that with the fact that the log data shows random numbers and he was at a loss on how to proceed with the bug solving. Only after actively helping him reproduce the bug locally he was able to solve it.
- **Bug 5**: Using the stack trace in the log event, Edwin was able to find the source of the bug without reproducing it himself. He did not need any log data to solve it.
- **Bug 6**: Edwin tried to find information in the log data after he discovered that the exception was not present in the log events. However, that information was randomized and not of any help. He concluded: "I would test locally if other databases contain the same error. If they do not contain the error then the error is specific to database 2018092 and not solvable without looking into it".

Edwin's opinion on the log component was positive. He was surprised that even though he did not see the real data, he learned that much from just the format of the data. There was one think he would like to have seen in the logs: consistency between IDs across different log data objects. During bug 4, Edwin opened the log data of two requests with the same URL. This means that the format of the log data is equal for those two requests. However, he was not able to confirm that those two requests regarded the same database object, since their IDs were anonymized to a different value. Although the anonymizer does anonymize equal values with equal paths to the same anonymized value, it would be nice to have that functionality across multiple log data to confirm that requests belonged to the same object.

B.3 Niek

Niek is a university student who has been working at *Het NIC* part time for 3 years. Of all developers, he is the most familiar with APRA. He is mostly a back end developer.

- **Bug 1**: Using the log data, he directly came to the conclusion that the view is not at fault here. While looking at the pieces of code which produces the error, he did not see the problem. That is why he viewed another piece of log data, the GET request after the POST. The log data of that GET request does contain the posted value, but as an integer. However, this did

not stand out to him. In the end, he needed to debug the problem himself to discover the problem.

- **Bug 2**: Niek found the correct logs fast and also quickly discovered that the problem was related to the double posting of the value `false`. However, it did not occur to him that it must be a front end problem at first, so he started looking at the back end. Perhaps this is due to him being mostly a back end developer. After he saw nothing out of the ordinary in the back end code, he found the problem in the front end.
- **Bug 3**: Where other developers stopped looking at the logs when they found the log data of the POST request, Niek looked further and wanted confirmation that the returned value was indeed 0, as the bug description says. He did find the value, but due to the anonymization method he could only confirm that the number was in the $[0, 9]$ range. First, he went to the back end, but seeing nothing odd there he found that the problem was in the view.
- **Bug 4**: Initially, he could not find the correct log events, due to him expecting another URL. After finding the logs, he quickly went into the back end without paying the log data any attention. Eventually he decided to reproduce the bug himself in order to fix it.
- **Bug 5**: Niek used the log data here to check if not some odd ID was being posted. After seeing that this was not the case, he quickly discovered the problem with the help of the exception stack trace.
- **Bug 6**: Accidentally, Niek already fixed this bug in the live version. He said: "this bug cannot be fixed by looking at the logs, since the background tasks do not log any exceptions". To fix this in a production environment, he would add proper logging to background tasks.

Niek's overall impression of the log component is good. He expressed that he is already a fan of it. He was surprised to see that it was possible to save some of the bugs without even starting his own local environment. His only remark was that he would have liked to see 0, instead of a number between $[0, 9]$, during bug 3. Then he could have used that as confirmation of the bug description.

B.4 Miquell

Miquell is a university student who has been working at *Het NIC* part time for more than 2 years. He has not worked on the statements process of APRA before.

- **Bug 1**: Instead of looking at the log data first, Miquell went to the back end. While that hunch is correct, he did not know where to look specifically. Seeing the data types in the log data, he started looking at the correct variables and discovered the problem quickly.
- **Bug 2**: After some searching, Miquell found the log events related to this bug. However, where the other developers drew conclusions from the log data, Miquell was more hesitant because he did not know whether the log events were related to the same object, due to the anonymization of the ID of the object. Therefore, he needed to locally reproduce the bug to fix it.
- **Bug 3**: Again Miquell went to the back end code first thing. After concluding that the back end looked fine, he took a look at the log data. There he discovered that the field was not being posted and was able to fix the view shortly after.

- **Bug 4**: The description of the bug suggests that the problem could be back end related, so that is where Miquell started. Not being able to find an error, he tried to look at the log data. However, he discovered that the logs would not help him. He needed to locally reproduce the bug to fix it.
- **Bug 5**: Miquell worked on this part on the system before and knows it by heart. He was able to navigate to and fix the code in minutes. He did not need any log events or data to solve this bug.
- **Bug 6**: Miquell asked if he was allowed to access the production database, since that is where the exceptions are stored that occur in background tasks. After being told he was not allowed to, he searched for logs that could help him. He was disappointed to discover that the exception message was anonymized. To solve this bug, he would add proper exception logging to the background tasks.

Miquell felt that the log data was only useful when he needed to fix a bug in a part of the system that he is less familiar with. Furthermore, he thinks the log data will be more useful when a bug is related of the parameters of a method call, as is the case with bug 1 for example. Similar to Edwin, Miquell would have liked if he was able to tell if different log data objects are about the same database object by anonymizing their IDs to the same value.

B.5 Sjors

Sjors is the only full time programmer working on APRA. He has been working for 2 years, so that makes him one of the most experienced ones with the system.

- **Bug 1**: Sjors solved this bug himself. Therefore, this bug was used to explain the new logging system to him.
- **Bug 2**: After finding the correct log data objects, he did not want to draw conclusions from the double `false` POST. This was because the IDs of the objects in the log data were anonymized and therefore he did not know if the POST request were related. His first guess was that the bug was caused by the back end because the format of the POST data looked fine. Sjors did not find any error there at a first glance so he started debugging the problem. Afterwards he said: "I could probably have solved it quicker if I could have confirmed the log data objects belonged to the same object".
- **Bug 3**: In contrast to the other developer, Sjors began to search much more efficiently for the correct logs. Filtering out unneeded log events and using a more specific time. Due to the missing value in the log data he reasoned that the error was related to the front end while mentioning that he was not completely certain. After taking a look at the view file he quickly saw that the field was missing.
- **Bug 4**: Again searching extensively, Sjors found the relevant log data quickly. The POST data showed a field called "Count". He wanted to know what that field did and went looking in the front end. He correctly reasoned that the problem was related to a difference between the "Count"-field and the refresh thereafter. However, the log data did not help him in exactly identifying the problem. So he needed to debug the problem to solve it, but was quicker due to knowing where to look.

- **Bug 5**: While searching for logs, he was looking at logs where a new user was created, while those logs were not related to the problem at all. He read the bug description again and then found the log event containing the error. However, he did not take a look at the stack trace. He went to the code and saw that the code could possibly throw some errors. Then Sjors finally looked at the stack trace and quickly identified the bug.

- **Bug 6**: Using his quick filtering technique, he concluded there was no error since the exception occurred in a background task. Having no access to the database, Sjors concluded that he could not solve the bug. He would add proper logging and then ask the user to reproduce the problem in order to solve the problem.

His overall impression was positive. he mentioned that it would need some learning from the team but that is no problem. He liked that he could follow users' action very detailed, he said: "it gives a more complete image of what the user has been doing around the time a bug occurred". The only thing he missed was consistency between different log data objects, as described in bug 2. He would have liked to know if the objects in the different log data were related.

B.6 Jesse

Jesse is a university student who has been working on APRA part time for around a year. He is mainly busy with back end related programming.

- **Bug 1**: The bug description does not mention which field the bug is about. This made Jesse conclude that there was nothing wrong with the log data itself. While he had a good hunch that the problem was related to conversion, he did not conclude that the back end is at fault. Eventually, he needed to reproduce the bug himself locally to fix it.

- **Bug 2**: Jesse failed to find the log events related to the bug. He did think that the values were not posted correctly, but that was because there were no log events according to him. After pointing him to the correct logs he did not draw conclusions from the log data. He went on to fix the bug be debugging it himself.

- **Bug 3**: The first thing Jesse did was looking at the data from the GET request after the POST request that adds the unknown credit note. Looking at the data, he concluded that the problem must be related to how the data was displayed. So he went to the corresponding view code but failed to find the bug, since the fault was in another view file. When he debugged, he saw that the field was missing on the view and was able to fix it.

- **Bug 4**: He found the right logs and immediately came to the conclusion that the log data was not useful in this case. His reasoning about the bug was excellent however and he found the problem quickly when debugging.

- **Bug 5**: Jesse forgot that the stack traces were visible in the log events. After I told him that stack traces can be found in the log events, he was able to fix the bug. At first he did not find the problem, since the line numbers in the stack trace slightly differ from the actual line numbers in the code. But since the exception also shows the type of the exception, he found the correct place.

- **Bug 6**: After finding the log data of the view that shows the error, Jesse was disappointed that the error message was anonymized. He explored the logs some more but finding nothing of interest, he said he would try to reproduce the error himself and otherwise add better exception logging.

To summarize, Jesse did not think the log data helped him much when solving the bugs. He prefers to debug the problems himself. He was happy with the stack traces in the logs however. As for missing information in the log data, he would have liked that the error message was not anonymized.

B.7 Slaven

Slaven is a consultant who works for *Het NIC* on various tasks for about 3 years now. He is also involved in APRA but generally does not work on the web application, which is what most bugs are about.

- **Bug 1**: Slaven started looking at the GET requests instead of the POST request. Due to the bug description he thought that the problem was in displaying. Therefore, he looked at the code that displayed it. After finding no direct problem with that, he reasoned that the problem could also be the POST. Since he did not find the log event directly related to this bug, he had to debug the problem to solve it.
- **Bug 2**: He filtered on a precise time instead of around the time the bug occurred, this was dangerous since he possibly missed logs. He found 1 related log event, but that was not enough to draw any conclusions. So he looked at the back end but did not spot anything wrong. He had to debug to discover the problem was related to the front end.
- **Bug 3**: This time, Slaven filtered correctly and then found the correct log quickly. It did not stand out to him that there was a missing field. After finding no error in the back end, he checked the view and discovered that the field was missing. This was without the help of the log data.
- **Bug 4**: This bug was skipped due to Slaven already knowing the solution.
- **Bug 5**: He quickly found the exception in the log data and shortly after found the error.
- **Bug 6**: Slaven found the correct log events and also looked at the log data of the error view. He was slightly disappointed to see that the error message was anonymized. After some searching in the back end, he gave up and said he needed the exception to know where to look.

Slaven's overall impression was positive. He did note that the log events and data would have been more useful in the first 3 bugs if he would have had more experience with the system and the web application code. He did like that he was able to see the path the user took through the application at the time the bug occurred.

C Risk assessment results

Table 9: Identifiable creditors per column combination per dataset

Combination of columns	**Amount of identifiable creditors**
Dataset A	**30.000 invoices**
$(address)$	5
$(InvoiceDescription, Address)$	4
$(IBAN, Address)$	1
$(InvoiceDescription, IBAN)$	1
$(InternalInvoiceNumber)$	3
$(InternalInvoiceNumber, InvoiceDescription)$	3
$(InternalInvoiceNumber, PaidAmount)$	3
$(InternalInvoiceNumber, InvoiceDescription, PaidAmount)$	3
$(InternalInvoiceNumber)$	3
$(DocType, InternalInvoiceNumber)$	3
$(DocTypeDescription, InternalInvoiceNumber)$	3
$(Journal, InternalInvoiceNumber)$	3
$(JournalDescription, InternalInvoiceNumber)$	3
$(DocType, DocTypeDescription, InternalInvoiceNumber)$	3
$(DocType, Journal, InternalInvoiceNumber)$	3
$(DocType, JournalDescription, InternalInvoiceNumber)$	3
$(DocTypeDescription, Journal, InternalInvoiceNumber)$	3
$(DocTypeDescription, JournalDescription, InternalInvoiceNumber)$	3
$(Journal, JournalDescription, InternalInvoiceNumber)$	3
$(DocType, DocTypeDescription, Journal, InternalInvoiceNumber)$	3
$(DocType, DocTypeDescription, JournalDescription, InternalInvoiceNumber)$	3
$(DocType, Journal, JournalDescription, InternalInvoiceNumber)$	3
$(DocTypeDescription, Journal, JournalDescription, InternalInvoiceNumber)$	3
$(DocType, DocTypeDescription, Journal, JournalDescription, InternalInvoiceNumber)$	3

Dataset B	**170.000 invoices**
$(InvoiceDescription, Address)$	1
$(InvoiceDescription, PostalCode)$	2
$(IBAN, PostalCode)$	2
$(InvoiceDescription, Address, PostalCode)$	1
$(InvoiceDescription, PaidAmount)$	1
$(InvoiceDescription, Currency, PaidAmount)$	1
Dataset C	**256.000 invoices**
Dataset D	**439.000 invoices**
$(InvoiceDescription, IBAN)$	8
$(InvoiceDescription, Address)$	2
$(InvoiceDescription, PostalCode)$	3
$(InvoiceDescription, City)$	1
$(Address, City)$	8
$(InvoiceDescription, IBAN, Address)$	1
$(InvoiceDescription, IBAN, PostalCode)$	1
$(InvoiceDescription, Address, PostalCode)$	2
$(InvoiceDescription, PostalCode, City)$	1
$(Address, PostalCode, City)$	8
$(InvoiceDescription, IBAN, Address, PostalCode)$	1
$(ExternalInvoiceNumber, InvoiceDescription)$	1
$(ExternalInvoiceNumber, PaidAmount)$	7
$(ExternalInvoiceNumber, InvoiceDescription, PaidAmount)$	1

D Algorithms

D.1 Number anonymization

[Algorithm redacted for publication]