ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 16

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

CREALOGIX E-Banking AG

Event Consolidation and Analysis Tool for E-Banking

by

Ueli Ehrbar

Supervised by

Prof. Dr. Gustavo Alonso (ETH Zurich)
Dr. Christoph Kuhn (CREALOGIX E-Banking AG)

June 7, 2011 – December 7, 2011

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The CREALOGIX E-Banking AG develops and distributes the CLX.E-Banking system, an online banking system used by several domestic and international banking institutions. The goal of this thesis was to design and implement two extensions to the system, a forensic analysis tool and an additional security system.

A forensic analysis tool is required to monitor the interactions between the e-banking system and its customers. It enables the system administrator to look at the details of each interaction, which is required to perform an investigation of an attack or any other fraudulent or suspicious behaviour that has been reported by a customer, a bank employee or a security system.

The system has an asynchronous background task collecting all available data, a filter mechanism discarding not required parts and a GUI integrated into the administration interface of the e-banking system that presents the details of each interaction to the administrator.

The additional security system is designed to analyse the behaviour of each customer and detect deviations in the behaviour that point towards an attack. The system describes the behaviour as a set of aspects, each consisting of several properties. The value of each property is constantly checked for its legitimacy. A set of rules determines if an illegitimate value is to be considered an attack, and if so, how the system should react.

The results of the thesis show the need to have such a security system. It is able to detect a number of different types of attacks, some of which could not be detected without it. The fact that this can be achieved without exhausting the system's full potential shows how powerful this approach to a security system is.

Acknowledgements

First and foremost I would like to thank Prof. Dr. Gustavo Alonso for granting me the chance of writing my master thesis outside of ETH Zurich. I understand this was an extra effort for which I am truly grateful.

I would also like to thank CREALOGIX E-Banking AG in the person of Dr. Christophe Kuhn for collaborating in this endeavour by giving me the opportunity and the trust to write my thesis in their facilities. Special thanks to Philipp Sieber, who explained the e-banking system to me and answered the questions I had.

Finally I would like to thank my family and my friends, especially Stefan Jucker, for the support throughout my studies and the master thesis.

Contents

1 Introduction

CLX.E-Banking [1] is an e-banking product used by many retail and private banks within Switzerland and abroad. Security and discretion are some of the most important attributes to a bank and should be represented by their e-banking product. To enforce these attributes, CLX.E-Banking provides a number of security related features. All these currently available features are either proactive or reactive. To this point no forensic functionality to analyse security related events exists. Such an analysis is designed to help understand the circumstances of the event and should result in an improvement of the security systems in place. In addition, the functionality should enable investigation of abuse not registered by the existing security systems but reported by a customer or bank employee. This may include identifying the owner of login credentials used to abuse an account with multiple login credentials or investigating an incident reported to the customer service.

1.1 Background

Several security components write log information in either classical log files or in database tables. The investigation and analysis are based on these log information. The most important sources are:

Audit Log Database The *Audit Log Database* contains all actions performed by the user within an e-banking session. An action is defined as a call to any web service or EJB. Each HTTP request from a user will trigger a number of such calls. Depending on the call, the log file contains information needed to trace back the call. This information includes the following:
- Information about the caller
- Time and date
- All parameters sent to the service (request)
- All information received from the service (response)

The audit log is available through the administration (web) interface of the banking application or through a direct database access (JDBC). Since this data resides within the same database as all the customer data, access will be very restricted. If the application is not running within the administration interface, this data will only be available by a report file.

Request Validator Database The *Request Validator Database* stores information about the sender of each HTTP request. This includes:
- Contract of the user
- Time and date
- All information sent in the HTTP request (e.g. URL, parameters, size)
- Information about the client system (browser name, OS name etc.)
- Information about the SSL session (SSL session ID etc.)

This information is stored in the same database as the *Audit Log Database*. It will not be possible to have a direct access to this database unless the application is running within the administration interface. Some detailed

information about the contents of this log file should remain anonymized for security reasons.

Request Validator The *Request Validator* compares metadata from the previous request (stored in the *Request Validator Database*) to the metadata from the current one. This is done by running the datasets through a definable set of rules. These rules can either be simple or complex. Simple rules compare the value of a parameter from the previous request to the value of the same parameter of the current request. A complex rule is the logical conjunction of any two rules. If a rule applies, one of the following actions will be taken:
- No action, if the rule is disabled.
- Logging the event.
- Freeze all payments created during this session.
- End the current session to force a re-authentication.
- Lock the contract. This prevents any future abuse of this account.

Note: This list is ordered. Each action involves all the previous actions. For example, if payments are frozen, the event is logged.

If a rule applies and an action is taken, the violation is written into a log file. This log file resides on the web server and can only be accessed by software running on the same server.

Web Application Firewall Log File CLX.E-Banking is running behind a *Web Application Firewall* (WAF). The WAF log contains security-related events. Relevant for this application are log messages related to blocked requests. Reasons for blocking requests include attempts of SQL injections, XSS [2] and CSRF [3].

1.2 Thesis Outline

The goal of this thesis is to design and implement *Forensic Analysis Functionality* and extend the collection of security related features with an *Anomaly Detection System*.

The *Forensic Analysis Functionality* combines security-related log entries from all available sources and analyses them. The findings of the analysis are visualized over a GUI in the CLX.E-Banking administration interface. They can be viewed, exported and extended with comments by the administrator.

The primary purpose is the investigation of fraudulent or suspect actions, after they have been logged by the security system or reported by a customer or bank employee. Targets of evaluation are:

Single e-banking sessions of an e-banking login. Example: Detection/analysis of a session hijacking.

Multiple e-banking sessions of the same e-banking login. Example: Detection of credential stealing.

Sessions of a group of e-banking logins, with access to shared money accounts (e.g. within a company). Example: Analyse activity on common money accounts (which payment was created by which login and when).

Optimization of security settings in regards of improving security (e.g. prevent false negatives in observed attacks) and customer satisfaction (e.g. prevent false positives which result in either the logout of a customer or the locking of an account).

The *Anomaly Detection System* analyses the customer's interaction with the system, tries to detect anomalies in it and react accordingly. The scope of the analysis is the whole interaction the customer has with the e-banking system. The detection of an anomaly invokes the same actions as the *Request Validator*.

2 Problem Analysis

The result of this thesis is a piece of software which increases the security of the CLX.E-Banking system. This requires an analysis of the potential threats to the system. To include presently unknown attacks, the analysis is based on attack vectors rather than on specific attacks currently known. For each identified attack vector the maximum harm of a successful attack and possible traces left in the system are identified. This information enables the successful identification of an attack to the system in retrospect.

2.1 Potential Points of Attacks

Currently, all of the 25 largest banks in Switzerland [4] offer web e-banking solutions [5]. They are commonly implemented as classic client-server web applications. This makes them vulnerable to the same attacks used on other web applications as described in the following.

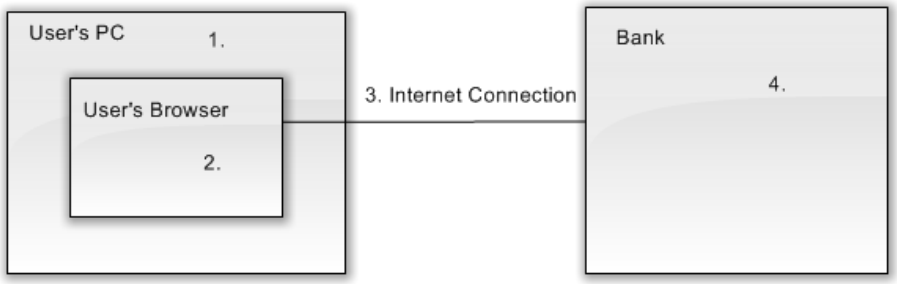

Figure 1: Possible attack targets

1. **User's PC** The system of a user can be compromised by a virus, a trojan or other malicious software. The need for each user to keep their anti virus software up to date and the sheer amount of users make this an easy target to an attacker [6].

2. **User's browser** The browser of the user can be infected. This includes the installation of malicious plug-ins or add-ons and changing browser settings such as adding a proxy controlled by an attacker. Browsers are designed to be easily extended and they are exposed to the most potential threats and are thus particularly susceptible to attackers. Therefore, attacking the browser is most likely the easiest way to get unauthorized access to an e-banking account.

3. **Connection over the internet** The connection between client and server can be attacked by, for example, man-in-the-middle attacks [7]. Such attacks are not to be considered a big risk unless an attacker can install malicious software or change settings on the user's PC.

4. **Bank infrastructure** The bank infrastructure is the most worthwhile target since it offers by far the largest return. It is centralized and controlled by the bank. This makes it possible to improve the security of the system

considerably and reduces the probability of a successful attack drastically. An attack would most likely be ineffective and therefore the bank infrastructure is an improbable target.

As described in chapter 1.2, this thesis will focus on attacks to the client-side. Thus attacks on the connection between client and server as well as attacks on the bank infrastructure will not be discussed any further.

2.2 Potential Attack Vectors

The potential damage of an attack to an e-banking system depends on the result rather than the technical means used to perform it. Hence attacks will not be classified by the technology necessary to perform them, but according to the kind of harm they can inflict on the system and any time constraints they might have. Attacks will be considered *online* if the successful execution relies on the attacked customer to be logged into the e-banking system. *Offline* attacks can be successfully executed at any time, regardless of the status of the attacked customer.

2.2.1 *Online* Attacks

Online attacks require an active e-banking session. The attacker can then use this session to get access to the customer's e-banking account. For these kinds of attacks, the attacker does not need any login credentials of the customer. *Online* attacks are usually based on a manipulation of the customer's browser or computer. Examples of such attacks include sending an asynchronous request to the server and tricking it into creating payments to bank accounts controlled by the hacker. This can be done by manipulation of the customer's browser or computer or by such means as cross-site scripting [2] or cross-site request forgery [3]. Another method would be to read the session identification (SID) used by the browser where the hacker can use the SID to access the session created upon the user logging in.

There are several strategies to prevent these types of attacks. They can be detected during execution, if the customer suddenly changes an aspect of its behavior or if the user has to utilize a second channel to confirm each payment. If the user is asked to confirm a payment he did not previously create, an attack has been detected. Second channels can either be fully separated from the internet, such as using SMS confirmation, or use a different connection to the bank. A different connection can be established by using additional hardware provided by the bank.

Ideally an attack should be detected and prevented automatically and as soon as possible. This prevents any mistakes the customer might make due to a lack of knowledge or carelessness.

2.2.2 *Offline* Attacks

Offline attacks do not rely on any concurrent user activity. The attacker needs to be in possession of the customer's login credentials to perform an *offline* attack. Thus, an attacker can then access the customer's e-banking account at any time. The method used by the attacker to gain access to the login credentials is insignificant for an attack to be classified as *offline*.

These types of attacks are commonly known and executed as either *phishing* [8, 9] or *pharming* [10]. *Phishing* tries to trick the user into disclosing his login credentials to the hacker. This can be done by sending an email containing a link to the homepage of the hacker. The homepage presents itself with the same look and feel as the e-banking system of the user. The user is requested to enter his login credentials, which instead will be sent to the attacker. *Phishing* attacks can only be prevented by reassuring the user that no bank will ever send such an email and that they should simply be ignored.

Pharming attacks forward the user to a hacker controlled homepage by changing some settings on the user's machine or browser. This can reach from changing the *hosts* file of the OS to simply changing the URL in the browser. Once the user is on this homepage, the attack works like a *phishing* one does. Like in a *phishing* attack, the infrastructure of the bank was not contacted at any point. Therefore, the bank has no technical means to prevent the user from revealing his login credentials.

To protect customers who have revealed their login credentials, the bank tries to detect suspicious customer behavior. This includes entering a payment to a suspicious bank account or geographic location. If a suspicious customer behavior is detected, the system reacts by locking the account. Performing a simple logout would not necessarily be sufficient, since the hacker could simply login again.

2.3 Security Systems in Place

To eliminate threats to the system, there are security systems already in place. As mentioned in chapter 1.2, these systems are additional features. Not every bank that uses the CLX.E-Banking also uses all the available features. Nonetheless, this analysis will describe all available features and the security they provide. The output generated by each one of these features is used as input for the application. Combining the output data of several different security features should offer a better understanding of an attack.

2.3.1 Web Application Firewall (WAF)

A *Web Application Firewall* is a proxy-based firewall. It provides security features to all web applications it protects. This way the implementation of these features for each application is unnecessary. Features include URL encryption and form protection. Encrypting an URL prevents manipulation of URLs. Asynchronous request to the server can not be done directly. The WAF only accepts URLs that have been encrypted and therefore have been sent to the user in a response to a previous request. This avoids accepting a request to paths not currently available to a user. Form protection is a way to validate the form fields and their content. It stores all form fields and their possible values for each response sent to the user. The system makes sure that no additional fields or invalid values are sent back from the user and if so, the system has detected an attack and does not forward anything to the web server. As a side effect, form protection is a way to avoid SQL injection attacks [11], where the hacker tries to execute some SQL code on the server. Such malicious requests are detected and will not be forwarded to the web server.

2.3.2 Request Validator

The *Request Validator* performs real time checks for each request. Each request is compared to the previous one and the differences are run against set configurable rules to detect any anomalies.

The *Request Validator* is designed to detect session stealing attacks. These are *online* attacks which try to open a new session with the same SID as the customer's session. This way the hacker shares a session with the customer and therefore has full access to the customer's e-banking account.

2.3.3 Payment Confirmation

To make sure that a hacker can not, independent of the means used to get access to a banking system, execute a payment undetected, the owner of the e-banking account has to confirm every payment over a second channel. This is an effective means to create additional security; any attacker would have to corrupt two channels simultaneously to get a payment executed. The second channel does not need the same capacity as the first one, since it's only used for confirmations. This allows the usage of second channels with small capacities like SMS.

To improve the usability of payment confirmation, features to exclude some payments are available. Two popular ones are bank account white lists and confirmation only for payments that exceed a certain amount of money. Bank account white lists allow the customer to create payments to accounts on the list without having to confirm them. This eases making periodic payments such as paying the phone bill or the rent. Payments not exceeding a predefined limit not requiring confirmation prevents the attacker to steal more than the predefined amount during a single attack. However, it is not clear why this would prevent any attacks over a longer period of time causing a similar or higher damage. The only advantage is a higher probability of detection and prevention of one of these attacks and a higher customer satisfaction due to the improved usability.

2.3.4 Hardened Browser

A hardened browser is a variation of a normal browser. It has several security related changes built in. These changes should prevent the attacker from gaining access to e-banking session, even if the hacker is able to install malicious software on the user's PC. A hardened browser is much harder to compromise than doing so with a normal one, since it does not allow any changes to it. Providing a hardened browser is a first step away from a web application towards a fat client. The bank supplies a separate browser to each client. This hardened browser can only be used in combination with the e-banking system of the issuing bank. Other than a fat client, the hardened browser is based on web technology and can access the web application. Therefore no fundamental changes to the server side are necessary.

2.4 Thesis Contribution

This thesis contributes to the security of the e-banking systems in two ways. One way is to extract as much information about an attack as possible. All the log files written by the security systems, as described in chapter 2.3, are

potential sources. The information gathered by the system can then be viewed by the system administrator and comments can be added. Beside the GUI needed to view the information, a report can be generated to summarize an attack. As a by-product, knowledge is gained about the attack vectors and how these attacks can be prevented. This knowledge can then be used to improve the configuration of the security systems in place or show the necessity of new ones. This part of the thesis improves the security of the system indirectly by providing a better understanding of the threats to the system. The other way is an additional security system. It is designed to profile each customer and detect abnormalities in his behavior that point towards an ongoing attack. This will actively improve the system security by interrupting previously unobservable attacks.

Configuring the security systems in place is done by a specialized external company. The same company will be responsible for the configuration of the results of this thesis. Therefore the result will be a framework with a reference implementation, where creating a long lasting configuration is not desirable since it has to change as the attacks change.

3 Forensic Analysis Functionality

This section describes the need for, the architectural choices and the implementation of the *Forensic Analysis Functionality*.

3.1 Introduction

Each time a bank became aware of an attack to their CLX.E-Banking system, they would manually have to gather all relevant security logs and send them to Crealogix E-Banking AG for analysis. A system developer would then have had to browse all the logs to collect the data associated with the event in question. From this set of data, conclusions were made and a report issued to the bank. This process is now simplified by installing the *Forensic Analysis Functionality* in the administration interface of the CLX.E-Banking system. The administrator is made aware of any attacks observed by a security system in place. The data is gathered automatically and presented to the administrator. If a customer of the bank reports an attack on his account, the administrator can check all interactions the customer in question had with the system and hopefully identify the fraudulent behavior. Additionally, if several customers have been the victim of the same/similar attack, then a report containing all attacked interactions can be generated.

The features of the *Forensic Analysis Functionality* are derived from this description. An asynchronous task periodically checks the logs for new entries. The new entries are parsed and run through a filter. The filter eliminates not security relevant entries, classifies the others according to the threat level the document and stores the data.

The administrator can monitor the data gathered by the asynchronous task over the GUI. The interaction of all customers can be searched and comments can be added to each entry of an interaction. Several interactions of different customers can be combined to an investigation and a report can be generated.

3.2 Architecture

The *Forensic Analysis Functionality* has to satisfy several constraints and meet the stated requirements.

The constraints arise from the integration into the CLX.E-Banking administration interface. The integration into the administration interface demands the usage of the same environment and technology. Both the administration interface and the e-banking system itself are implemented in Java and run in a WebLogic EJB container [12]. The GUI is implemented using servlets that generate XML output which is transformed to XHTML by the use of XSLT files.

The requirements to the *Forensic Analysis Funtionality* are met by implementing it as a extendable framework and a reference implementation. This section will concentrate on the concepts used to design the framework. Detailed information about the reference implementation can be found in section 3.3.

The system can be extended by adding new data sources or new filters. This extensibility forces the framework to offer concepts abstract enough to allow all imaginable applications. This is done by applying separation of concerns and providing narrow interfaces. The separation of concerns leads to a partitioning of

the funtionality into several different modules. Each module, and the interfaces it provides, is described in the following.

3.2.1 Modules

This is an overview over the separation of concerns and its influence on the system design. Each of these five modules handles a different aspect of the project which is described in detail in the following.

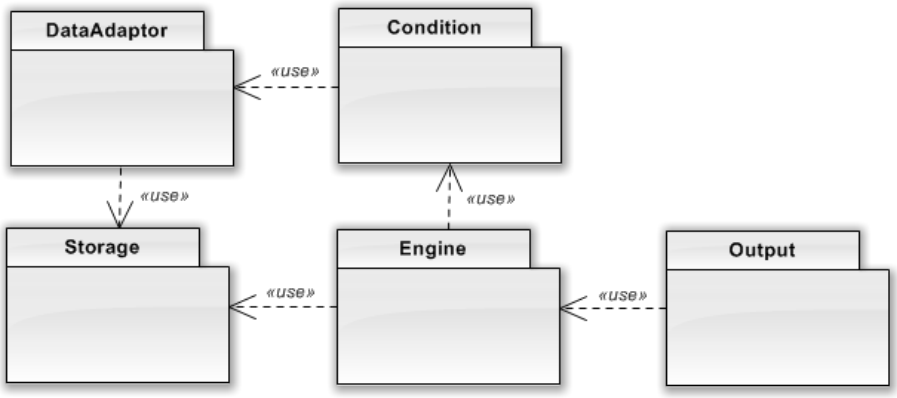

Figure 2: Module overview

DataAdaptor The application depends on data from several heterogeneous data sources. Each of these types of data source is accessible over a specific adaptor with a uniform interface. This ensures the decoupling of each data source and the application. It also presents an easy way to extend the application with an additional type of data source.

Each *DataAdaptor* is specific to a type of source and it provides the content of its source to the application. The data is presented as a uniform log entry, which can be a row in a database or a line in a log file. This enables the data adaptors as well as the data sources they access to be interchangeable.

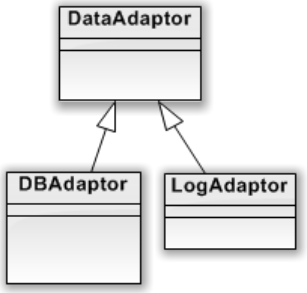

Figure 3: The abstract *DataAdaptor* with two implementations, one to read from a database and one to read from a log file

Condition A *Condition* is a filter. It is used to either discard or classify each entry read from a log. A log entry is discarded if no *Condition* applies, or classified otherwise. The classification of a log entry equals the highest classification of an applying *Condition*. This makes the *Conditions* a crucial part of the application. Depending on the *Conditions* in place, an attack is detected and classified accordingly or the log entry in question is discarded.

A *CombinedCondition* allows a logical combination of any two *Conditions*. This allows the reuse and the combination of existing *Conditions* and makes the transformation of complex algorithms to a *Condition* easier because it allows the creation of more complex structures. The support of arbitrarily complex algorithms is necessary since the detection of an attack might require one.

Note: *Combined Conditions* will usually use the logical *AND*. The usage of another logical operator is not necessarily meaningful since it can usually be replaced by creating two separate *Conditions*.

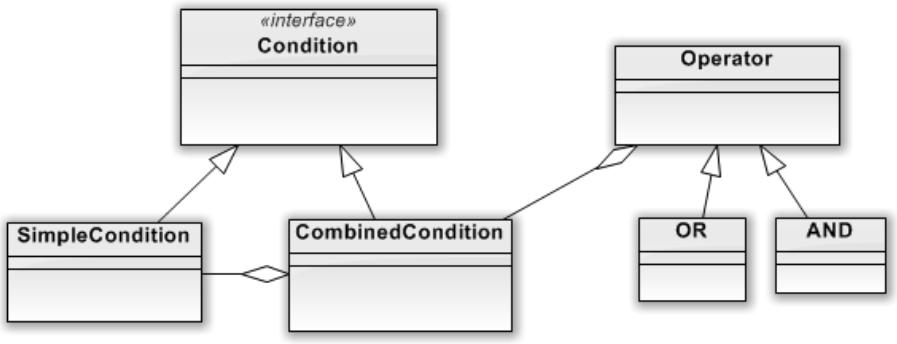

Figure 4: *Condition*, *SimpleCondition* and *CombinedCondition* with the corresponding *Operators*

Storage The *Storage* module contains the following three data representation objects and no business logic.

The *Entry* object is the logical and uniform representation of any log entry. This is the abstraction needed to be able to handle log entries from any source. For each *DataSource*, a subtype of *Entry* can be defined to deal with the specifics of each log. These subtypes of *Entry* are also part of this module.

The *EBankingSession* is a collection of all *Entries* generated by one customer during one interaction with the system. An interaction usually starts with a login and ends at the logout.

A *Case* is not part of the data collection process. It is used to associate several *EBankingSession* to a specific problem or event. This functionality is available in the GUI of the administrator interface.

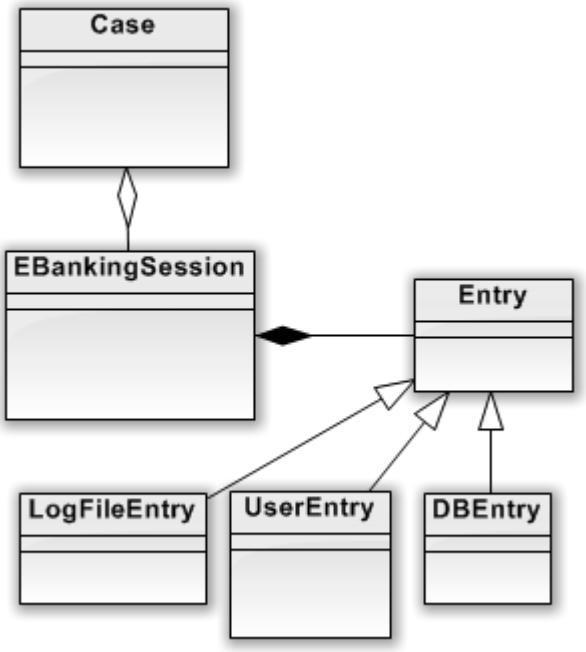

Figure 5: Storage module containing the data representation objects and their implementation

Engine The *Engine* uses all the functionality described above. It does this by collecting the newly available *Entries* from the *DataAdaptors* and filters them using the *Conditions*. All *Entries* not matching at least one *Condition* are discarded, and the rest is assigned to the respective *EBankingSession*. The *EBankingSessions* are managed by the *Engine* and upon reading the first *Entry* of a contract, a new one is created. All successive *Entries* generated in the interaction with the same contract are assigned to it. The *Engine* then closes the *EBankingSession* if no *Entry* was assigned to it during a certain period of time. This timeout is needed to be able to examine the success of the logout procedure.

Output The *Output* module is integrated into the environment of the administration interface of the CLX.E-Banking system. It consists of servlets that generate XML output and XSLT files that generate XHTML. The intermediate level of generating XML enables a quick replacement of the presentation layer without any changes in the servlet layer. The GUI is designed to monitor the results of the *Forensic Analysis Functionality* and work with them. The configuration can not be changed over the GUI.

3.2.2 Data Flow Analysis

The application distinguishes between operational data and the configuration data. All settings and configurations for the application itself are considered

configuration data. This small amount of data is handled separately and the data flow analysis will focus on the operational data.

The amount of operational data that has to be handled by the *Forensic Analysis Functionality* grows as the number of customers using the CLX.E-Banking increases. Therefore, it is crucial to filter and discard all data that is not needed to reconstruct any security related events. The necessity to keep the data is assessed by the *Conditions*. This step should eliminate a big percentage of the data. The remaining part is stored separately for any later use. This has to be done continuously, since most of the data sources used are available only temporary. Log files, for example, are copied from the operational server and stored on a mass storage server where they are not accessible anymore.

These constraints are implemented in the *Forensic Analysis Functionality*. Each *DataAdaptor* collects all available data from its source. The collection of all data is required to avoid missing important data. This bulk of data is run through the set of *Conditions*. This is the bottleneck of the *Forensic Analysis Functionality*. On one hand, the set of *Conditions* and their complexity are important to make an informed choice about the importance of each entry, on the other hand, each additional *Condition* slows down the whole data elimination process. With an increasing load to the systems, both the CLX.E-Banking system and the *Forensic Analysis Functionality*, the set and the complexity of the *Conditions* has to be tightened.

After the elimination process, the remaining *Entries* are stored in the database. If the set of *Conditions* diminish the number of *Entries* in sufficient way, neither storing them in the database nor the space used in the database will be of significance.

The impact of running the *Forensic Analysis Functionality* on the performance of the whole system should not profound. The total number of log entries to be processed by the *Forensic Analysis Functionality* is equal to the number of log entries written by the CLX.E-Banking system. Thus the load increase to be handled by the *Forensic Analysis Functionality* will at most be equal to the load increase experienced by the CLX.E-Banking system.

3.3 Implementation

Most of the *Forensic Analysis Functionality* is realized as a stand-alone Java project. The GUI has to be integrated into the administration interface GUI project of the CLX.E-Banking system.

In the following, the implementation of the application as a whole and the specifics of each module are described. The objects not mentioned in section 3.2 are part of the reference implementation and not the framework.

3.3.1 DataAdaptor

The *DataAdaptor* has a simple interface. It allows the *Engine* to check if new updates are available, using the *entriesAvailable* method, and to retrieve them using the *getNextEntries* method. During the execution of the *getNextEntries* method new entries might be read. They will be made available after the current call is finished.

Two different subtypes of *DataAdaptor* are defined:

DBAdaptor As the name suggests, the *DBAdaptor* is designed to access the logs stored in a database. It is a generic implementation that should work on every table of a given data source. Three values are needed for the configuration: a statement to determine if there are new records in the database, a statement to select the new records and the type of *Entry* to store each record in.

LogFileAdaptor The *LogFileAdaptor* accesses a log file. The file size is the indicator used to determine the presence of new log entries. These are read and stored as the configured type of *Entry*. Besides the type of *Entry*, the path to the log file has to be specified in order for the *LogFileAdaptor* to run.

For any additional type of log, a new subtype of *DataAdaptor* has to be implemented.

3.3.2 Condition

A *Condition* is used to decide whether to keep or discard a certain *Entry*. This is done using the *applies* method. It takes an *Entry* and returns *true*, if and only if the *Condition* is met by the *Entry*. Additionally, each *Entry* that meets at least one *Condition* invokes an action in the system. The possible actions are described in section 1.1. An action can be configured for each *Condition* and is called *Classification*. In addition, each *Conditions* has an active flag. It is used to distinguish between active *Conditions* and *Conditions* that are only used as part of a *CombinedCondition*. These two values are the only configuration needed for the generic type of *Condition*.

Each subtype may require additional configuration. The list of provided subtypes is:

CombinedCondition and SimpleCondition These two objects have a special purpose. The *CombinedCondition* allows the conjunction of any two *Conditions*. The two sub-*Conditions* have to be configured to be able to use the *CombinedCondition*. The *SimpleCondition* however offers no additional functionality. It is only used to classify the type of *Condition* implemented.

PropertyValueCondition This *Condition* checks if any given *Entry* contains a property with a given value. It needs the name of the property, the expected value and a boolean value that specifies if the presence of the value is expected (i.e. it can check if an *Entry* does or does not contain a certain value for a property).

A variation of this *Condition* is the *PropertyValueContainsCondition*. Rather than checking a property for equality, it checks for substrings matching or not matching the given value.

EntryFromLogCondition As the name suggests, this *Condition* is designed to check if an *Entry* is of a specific type. This is done by comparing the class name of the given *Entry* with the one provided in the configuration. This *Condition* is usually used as part of a *CombinedCondition*.

Using the *Conditions* from this list a variety of tests can be performed. Due to their importance to the effectiveness of the system, additional *Conditions* and more complex ones would be desirable.

3.3.3 Storage

The *Storage* module stores all collected information in the database. The GUI can then load this information from the database and present it to the administrator. Due to the nature of the data, there is neither the necessity nor the possibility to edit it. The only mutable data is created by the administrator over the GUI, namely the information stored in a *Case*. This includes the name and comment of each *Case* as well as the *EBankingSessions* assigned to each one.

To make the extension of the system as easy as possible, the interaction with the database is handled generically. Each *Entry* is serialized and stored as an object.

The database is accessed over a data pool provided by WebLogic [12]. The database implementation does therefore not play a role. The *Forensic Analysis Functionality* introduces four new tables:

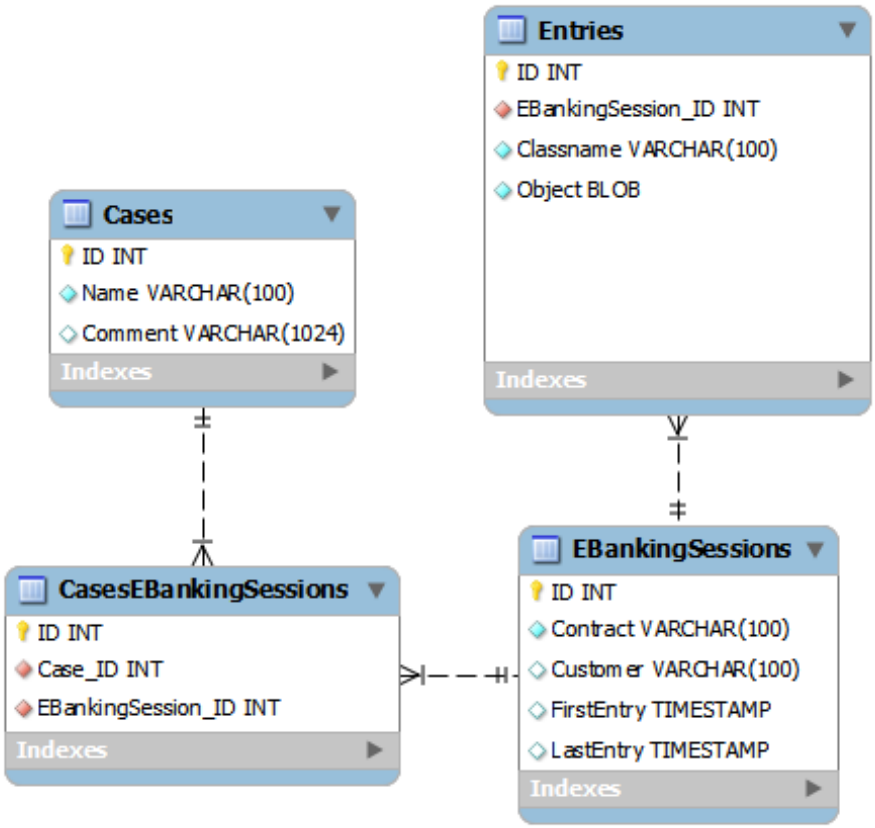

Figure 6: Database schema for the *Forensic Analysis Funtionality*

Entries Stores the entire *Entry* objects with some additional ordering information. It holds the foreign key of the *EBankingSessions* because an *Entry* can only be assigned to one *EBankingSession*.

EBankingSessions The value of both the *FirstEntry* and *LastEntry* column can be looked up by browsing all *Entries* assigned to the *EBankingSession*. They are stored as an attribute of the *EBankingSessions* table to avoid repeated queries to determine their value. The *Contract* is needed to link log entries to a customer. It is the primary identification of any customer of the CLX.E-Banking system.

CasesEBankingSessions This table is used to represent the many-to-many relationship between a *Case* and an *EBankingSession*.

Cases The content of this table is created by the administrator over the GUI. It represents a grouping of *EBankingSessions* and consists only of a *Name* and a *Comment* field. The content of this table as well as the assignments of the *EBankingSessions* can be deleted by the administrator.

Additional to the fields described above each table has an auto-increment primary-key field called *ID*.

3.3.4 Engine

The only content of the *Engine* module is the implementation of the *Engine*. It implements the *java.util.Runnable* interface since it has to be run asynchronously. The functionality described in section 3.2.1 is implemented in the *run* method derived from the interface.

The *Engine* needs little configuration. It needs the timeout after which the *EBankingSessions* are closed, a list of *DataAdaptors* and a list of *Conditions*. Additionally, the time between two executions of the *run* method can be configured. This optional value of type long is set to one second by default. This can be overwritten by providing another value for the property *sleep* in the configuration.

3.3.5 Output

This module is integrated into the GUI project of the administration interface and implemented accordingly. It has a single point of entry accessible like every other page of the administration interface and presents itself with the same look and feel.

As described in section 3.2.1, the GUI is implemented using servlets that generate XML output. This is transformed to XHTML by an XSLT file. The servlets can access the backend system using service calls. The system offers an interface where all available backend services can be called. The caller has to provide the name of the service and an XML document specifying all parameters. The services added to the system for the *Forensic Analysis Functionality* are implemented in a single Java class which itself can accesses the classes from the *Storage* module described in section 3.3.3.

This is the complete list of implemented services:

addEBankingSessionToCase Adds a *EBankingSession* to a *Case* by inserting a new line into the *CasesEBankingSessions* table.

addUserEntry Stores a new *UserEntry* in the corresponding *EBankingSession*.

createNewCase Creates a new *Case* and stores it in the database.

deleteCase Deletes a *Case*.

getAssociatedContracts Returns a distinct list containing the contracts of *EBankingSessions* assigned to this *Case*.

getCaseById Returns the *Case* with the specified id.

getEBankingSession Returns the *EBankingSession* with the specified id.

getEBankingSessionsByCase Returns all *EBankingSessions* assigned to a given *Case*.

getEBankingSessionsByContract Returns all *EBankingSessions* of a given contract.

getReport Generates the report for a *Case*.

removeEBankingSessionFromCase Deletes the association of an *EBankingSessions* with a *Case*.

searchCases Returns a list of *Cases* matching the search criterion.

searchContracts Returns a list of contracts matching the search criterion.

searchCustomers Returns a list of customers matching the search criterion.

Note: All of these services have an identical parameter list. Each requires the request, response and the environment object of the service. The request object contains the values needed to execute each service. The return value of the service is stored in the response object. The service environment is mainly used for the exception handling.

The most important part for the administrator is the GUI integrated into the administration interface. It consists of several different pages described in the following.

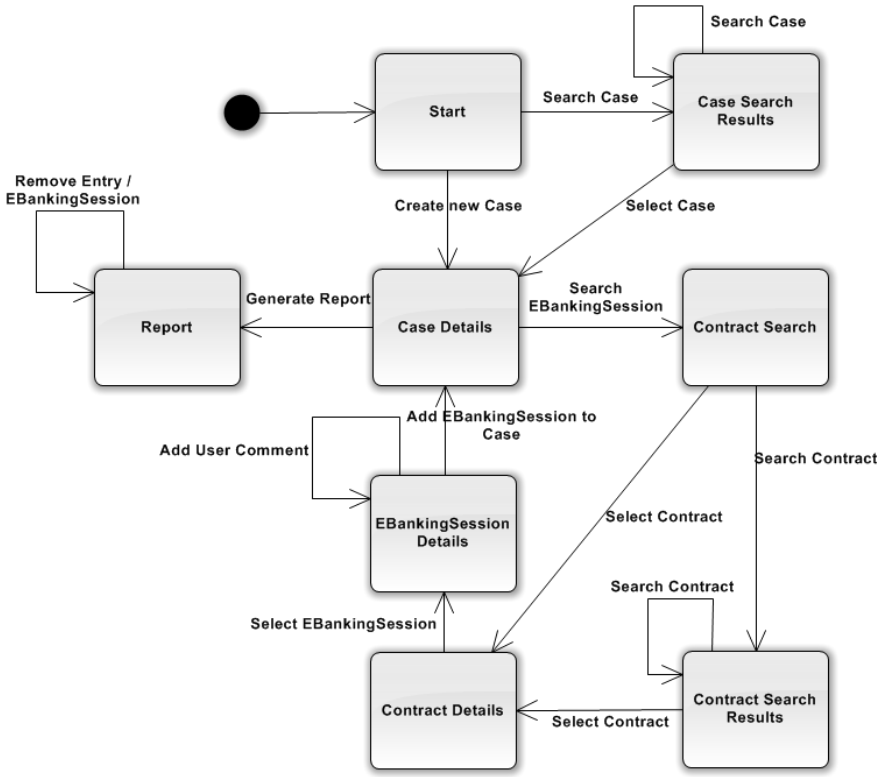

Figure 7: Sitemap

Start This page is opened over the navigation of the administration interface. It offers the possibility to search for existing *Cases* or create a new one by inserting its name and an optional comment.

Case Search Result The title and the comment of the *Cases* matching the search criterion are displayed here. To ease any additional searches, this page offers the same search functionality as the *Start* page does.

Case Details The details, such as the date of the first and last *Entry*, of each *EBankingSessions* assigned to a *Case* are displayed here. The page offers a link to the detail view of each assigned *EBankingSession* as well as a remove button for each one.

This page is essential to the whole process. It displays the *Case* currently under examination and all following pages offer operations on it.

Contract Search This is the first step in adding an *EBankingSession* to a *Case*. To find the *EBankingSession*, a contract has to be searched or selected. To search a contract, the administrator can perform a full text search. If the contract in question has already one of its *EBankingSessions* assigned to the same *Case*, it can be selected from a drop down. This way the contract details can be displayed directly without the detour over the search results.

Contract Search Result The results of the contract search are displayed on this page. Additionally, like on the *Case Search Results* page, the search string can be changed and a new search can be performed.

Contract Details The contract details consist of all *EBankingSessions* on record for a given contract. For usability reasons only the *EBankingSessions* from the previous month are displayed by default. The displayed time period can be changed using two date fields displayed on the page.

Note: If the owner of the current contract is concurrently logged in to the system, the concurrent *EBankingSessions* will be displayed by default. Since it is concurrent, no *LastEntry* date can be displayed.

EBankingSession Details All *Entries* currently assigned to the selected *EBankingSession* are displayed here. This list contains a visualization of the classification, the timestamp and the summary of each *Entry*. This page offers the functionality to add the *EBankingSession* to the current *Case* and to add a comment to the displayed *EBankingSession*.

A comment can be added by selecting the existing *Entry*. The title and the content of the comment can then be entered on the bottom of the page, where the selected *Entry* is shown again. Upon saving the comment, the list is reloaded and contains the user comment next to the previously selected *Entry*. The user comment is stored as an *UserEntry*, as described in section 3.3.3. From this time forward the user comment is part of the *EBankingSession*.

In addition, this page is able to track the customer's interaction with the system that are concurrently in process. This can be done by simply refreshing the page. The tracking delay is equal to the amount of time the system requires to write the log data, read it again using the *Forensic Analysis Functionality* and store it in the database. This time usually is in the order of a few seconds.

Note: An *EBankingSession* can only be added to a *Case* once. If the *EBankingSession* in question is already part of the *Case*, it can not be added anymore.

Report The report sums up the entire content of a *Case*. It consist of a list of all *EBankingSessions* associated with the *Case* and all their *Entries*. The selection of the important *Entries* or *EBankingSessions* can be done by removing the other ones. The removal of an *EBankingSession* removes all its *Entries* too. The removal of any content of the report will only influence the report, but not anything else.

The content displayed for each *Entry* is equal to the *EBankingSession Detail* page.

3.3.6 Configuration and Language Localization

The integration of the *Forensic Analysis Functionality* into the CLX.E-Banking system demands the implementation of the concepts used for both configuration and language localization. Especially the configuration comes with some additional constraints which are described in this section.

The CLX.E-Banking system can be configured using the Spring Framework [13]. It is based on a XML file containing the description of all beans that need configuration. To run the *Forensic Analysis Functionality*, some configuration is needed. The *Engine* itself needs the timeout after which a *EBankingSession* has to be closed (see section 3.3.4). Additionally, it requires the list of the *DataAdaptors* to collect *Entries* from and the list of active *Conditions* to filter that list. The configuration needed for each subtype of *DataAdaptor* or *Condition* can differ and explained in detail in section 3.3.1 or 3.3.2 respectively.

The Spring Framework can return an initialized instance of every bean defined in the underlying configuration file. If changes to a bean in said file are made, the file can be reloaded and the new instances of the bean will have the adapted configuration. This mechanism is used to apply configuration changes in the *Forensic Analysis Functionality*.

The architecture of the *Forensic Analysis Functionality* allows the replacing of the configuration during runtime. This can be done by replacing the *Engine* object. The changes have to be made in the XML file and reloaded in the Java context. A new instance of the *Engine* can be loaded from the new configuration.

To make sure no content of any log is missed, the old instance has to be stopped, replaced with the new instance and started again in an atomic operation. Content of a log can only be missed if these steps are not performed as one atomic operation.

The GUI is the only part of the *Forensic Analysis Functionality* that is subject to language localization. The CLX.E-Banking system has an XML based solution. Each string displayed on the GUI has a key. The corresponding value is stored in a language specific XML file. Each file defines the translation of each key for one language. The language on the GUI can be altering by changing the underlying XML file.

4 Anomaly Detection

The *Anomaly Detection System* is an additional security system for the CLX.E-Banking system. It is described in the following.

4.1 Background and Environment

This section describes the requirements to be met by a security system and the environment the security system and the e-banking system are in.

A security system not only needs a way to detect attacks but also effective means to counter them. As described in section 2.2, attacks to an e-banking system usually aim to transfer money. An effective countermeasure prevents this from happening. Such countermeasures are implemented in the CLX.E-Banking system, as described in section 1.1, and used by the *Anomaly Detection System*.

An e-banking system is an application that uses the functionality provided by the core banking system, which manages the business conducted by the bank with its customers. This includes handling deposit accounts, payments, mortgages and loans. Each bank runs its own core banking system. To prevent abuse of the core banking system, it has its own security system in place. Any payments entered in an e-banking system are transferred to the core system where they will be executed. Once a payment is transferred, the e-banking system can not influence it anymore.

This limits the capabilities of an e-banking system to prevent attacks. They have to be detected and prevented before they reach the core banking system. This makes the timing between the detection of an attack and the transfer of a payment to the core banking system critical. To make sure a security system is effective, this timing has to be analysed. For the *Anomaly Detection System* this is done in section 4.4.7.

4.2 Introduction

The security systems in place are designed to prevent several types of attacks. Not amongst these attacks are *offline* attacks as described in section 2.2.2.

The detection of an *offline* attack is only possible if the attacker logs in using previously stolen login credentials. This is the only interaction the attacker has with the e-banking system. Since the attacker is able to authenticate himself as a customer, access to the system will be granted. By simply stealing the login credentials of the customer, the attacker is not able to imitate the customer's behaviour.

The *Anomaly Detection System* tries to exploit these behavioural differences between the customer and his impersonator by analysing the customer's behaviour and detecting any anomalies in it [14]. To be able to do so, the system analyses all interaction it has with each customer and extracts the expected behaviour from it. Each time a customer logs in, his behaviour is tracked and constantly compared to the expected one. If any anomalies are detected, countermeasures are invoked. Otherwise the current interaction is included in the expected behaviour and stored in the database.

4.3 Architecture

The analysis of the customer behaviour is very complex. To master this, the *Anomaly Detection System* uses a kind of divide and conquer approach. The customer's behaviour is described as the union of several different behavioural aspects. Each aspect deals with a specific property of the customer's behaviour, the geographic location for example. By limiting an aspect to a certain set of behavioural properties, changes in the aspect can be detected easier. To make the system more flexible, the detection of an anomaly in a single aspect does not necessarily invoke a reaction of the system. This allows the system to tolerate, for example, a travelling customer logging in from a different location than usual.

The architecture has to address the core problem of extracting the customer behaviour in a way that allows storage and comparison. This extraction has to work both for long term data, such as the entire interaction between the e-banking system and the customer, and short term data such as the current interaction. Furthermore, the structure has to offer a way to include short term data into the long term data.

The integration of the *Anomaly Detecion System* into the CLX.E-Banking system forces the same constraints on the system than the integration of any other subsystem. Each system has to run on a WebLogic [12] EJB container and has therefore to be written in Java. In addition to these constraints, the *Anomaly Detection System* has to be integrated into an environment that can provide the information required to run the behavioural analysis of the interaction the system has with the customers. The minimum information required is all interaction the system has with each customer. While parts of this information are available in various subsystems of the CLX.E-Banking, the *Forensic Analysis Functionality* has a the ability to collect all information available in the system. The sources of the information required to perform the analysis are already sources of the *Forensic Analysis Functionality*. This makes the integration of the *Anomaly Detection System* into the *Forensic Analysis Functionality* very useful.

Similar to the *Forensic Analysis Functionality*, the *Anomaly Detection System* is designed as an extendable framework and a reference implementation. By doing so, the framework can be extended to detect previously undetectable anomalies.

4.3.1 Solution Concept

The solution is based on a structure holding all information about the interaction of a single customer and the system. This structure is stored in the database and loaded when the corresponding customer logs in. It represents the long term data used to describe the expected behaviour of each customer. To allow some changes in the customer's behaviour, a set of rules determines which changes will result in a reaction of the system and which changes will affect the expected behaviour, which of course is updated in the database if it changes. Each rule may include several aspects and each aspect may be used by several rules.

An aspect is defined by a set of behavioural properties. For each property, the aspect holds the expected value. If the current value of a property is not accepted, a difference has been detected. If a rule determines that this difference

is significant, it is considered an anomaly. Otherwise the new value is added to the accepted ones.

While the *Anomaly Detection System* is integrated into the *Forensic Analysis Functionality*, it is a system on its own and it is designed that way. The whole functionality of the *Anomaly Detection System* is encapsulated into a single object. This encapsulation separates it from the surrounding system, usually the *Forensic Analysis Functionality*. However, it is designed to work in any environment that can provide the information required for the detection of anomalies.

4.3.2 Object Model

The object model contains the most important objects for the *Anomaly Detection System* and is described in the following.

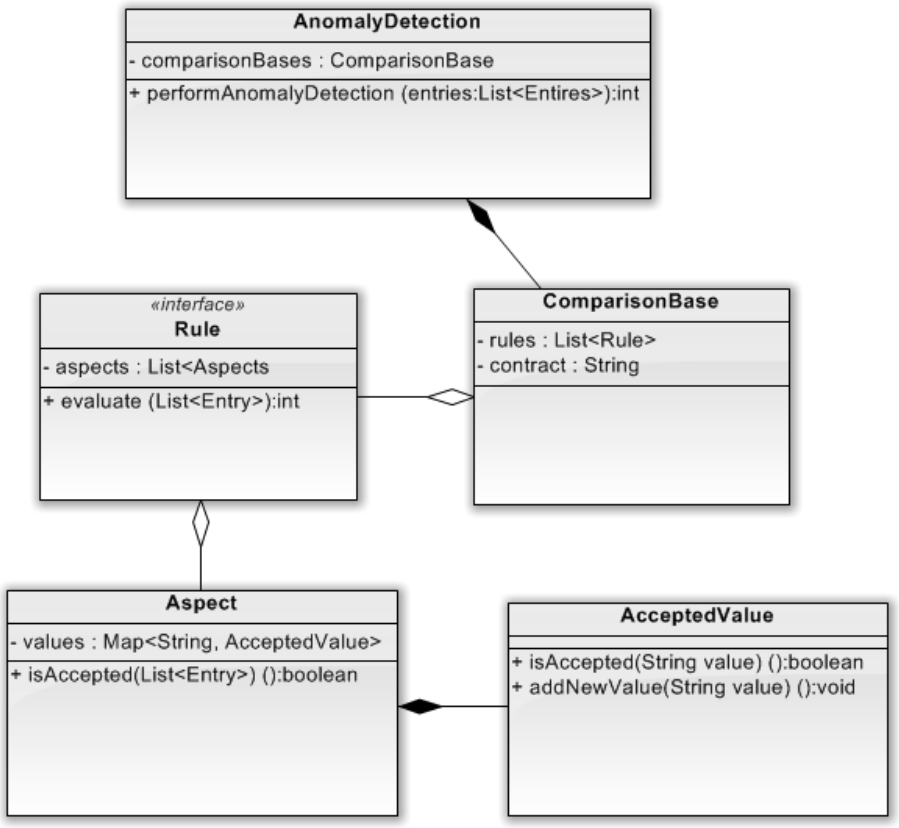

Figure 8: The object model of the solution concept

Aspect and AcceptedValue An *Aspect* represents a single aspect in the customer's behaviour. It consists of a list of properties describing that behavioural aspect.

In this context, a property is a single value from a single *Entry* of any

data source that is under surveillance. The set of possible properties of each *Aspect* has no constraints. This means that properties from several or all available data sources can be used to describe a single *Aspect*.

For each property, a corresponding *AcceptedValue* is defined. While the corresponding value of a property is a *java.lang.String*, more complex operations than a simple check for equality will be required in most cases. The *AcceptedValue* is the abstraction needed to provide the flexibility to do so. Any arbitrarily complex algorithm can be implemented as a subtype of the *AcceptedValue* object. This additional abstraction allows to have a more fine-grained system by enabling grey areas in the assessment of the value. This is done by assigning a classification to each not accepted value rather than just a boolean. If an unaccepted value is a clear indication of an impersonated identity, a higher classification can be assigned, and if the unaccepted value is a small irregularity, it can be assigned a low classification.

In order to provide a better understanding, here is an example. An easily observable and checkable aspect of the customer's behaviour is the browser used to log into the e-banking system. Therefore an *Aspect* representing the browser is configured. This is done by collecting all properties from any sources that allows to make a conclusion about the browser. They can be freely combined to describe an *Aspect* as accurate as possible. To keep the example simple, the *Aspect* will consist of a single property that contains the name of the browser. An *AcceptedValue* that allows only a single value is assigned to the property.

Upon the first login of the customer, the *Aspect* will read the name of the browser and pass it on to the *AcceptedValue*. The *AcceptedValue* will check if the current browser name is equal to the previous one. Since this is the first login of the customer, the *AcceptedValue* has not stored a previous browser name and will therefore store the current one. In all further interactions the customer has with the system, the name of the browser has to match the current one. If this is not the case, the *AcceptedValue* assigns a classification to the current value to assess the magnitude of the difference. If, to continue the example, the browser name does not match anymore because the name has changes due to an software update, a lower classification can be assigned compared to a complete change of browser name.

While this is a very simple example, it shows the usage of these two key concepts. However, the simplicity does not show another advantage of the *AcceptedValue*. Depending on the complexity of the *Aspect*, a simple check for equality might not be sufficient. A list of values, a generic expression or a mathematical formula might be required to check the validity of a value. The *AcceptedValue* is designed as a seperate object to allow this to be done by subtyping.

Rule A *Rule* contains a list of *Aspects*. It is designed to determine if any differences detected in the customer's behaviour by any of its *Aspects* qualifies as an anomaly and if so, returns the code of the appropriate action to be invoked by the system. This decision can be based on the amount and

the classification of the detected differences or on the differences themselvs. Under this circumstances a standard implementation of a *Rule* is not meaningful. Therefore the *Rule* is designed as an interface in the framework and a preliminary implementation is part of the reference implementation of the *Anomaly Detection System*.

ComparisonBase The *ComparisonBase* is the abstraction of all interaction a single customer had and is currently having with the system. It is specific to a single contract and contains the list of *Rules* to be evaluated. The result of an evaluation is the highest action code returned from a *Rule*.

AnomalyDetection The *AnomalyDetection* object is the interface of the surrounding system. It encapsulates and executes the functionality of the *Anomaly Detection System*, such as invoking the action corresponding with the return code of the *ComparisonBase*. It can be integrated if all necessary information and an instance of the *AnomalyDetection* are present.

4.4 Implementation

This sections describes the implementation of the *Anomaly Detection System* and its integration into the *Forensic Analysis Functionality*.

4.4.1 Integration into the Forensic Analysis Functionality

The integration of into the *Forensic Analysis Functionality* offers several advantages as described in section 4.3. In addition to them, the integration allows the reuse of several modules of the *Forensic Analysis Functionality*. The reused of the modules and the extensions to them are described in detail in the following.

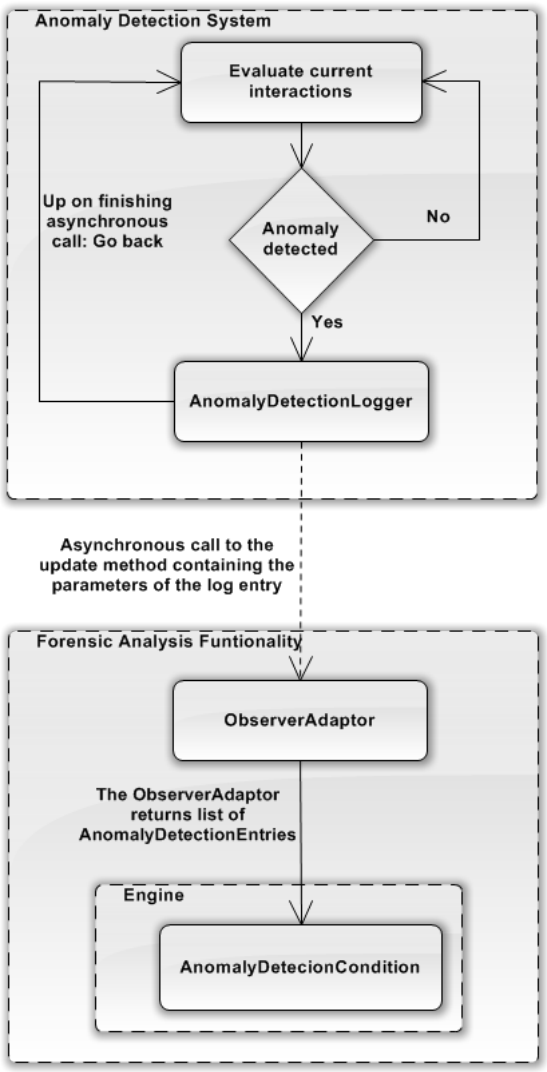

Figure 9: Integration of the *Anomaly Detection System* into the *Forensic Analysis Functionality*, using the mechanism designated to integrate a security system

Storage The *Storage* module described in section 3.3.3 is used to represent, store and load the data. The data required by both systems is very similar, both need access to certain parameters of an *Entry*. The interface of the *Entry* is designed to allow access to any kind of underlying log data, as described in section 3.2.1. Since the analysis performed by the system relies on data from logs, the *Entry* is very suitable by design and does not require modification to be used. In addition, the *Storage* module does not contain any business logic but is merely used for data representation. For these reasons the *Storage* module is used as data representation in the *Anomaly Detection System* as well.

Output The *Anomaly Detection* is another security system for the CLX.E-

Banking system. As such, it generates log output using the *AnomalyDetectionLogger*. The *AnomalyDetectionLogger* is a simple logger that encapsulates all logging related features required by the system. It is designed to be extended according the requirements each bank has on the log output of the security features of the CLX.E-Banking system once both the *Forensic Analysis Functionality* and the *Anomaly Detection System* are put to use. Its most important function as of now is to forward the log messages from the *Anomaly Detection System* to the *Forensic Analysis Functionality*.

The fact that the *Anomaly Detection System* is an additional security system makes its output an important source for the *Forensic Analysis Functionality*. Therefore an additional *DataAdaptor* as described in section 3.3.1 has been implemented and is described in the following.

The *ObserverAdaptor* is a subtype of the *DataAdaptor* and implements the *java.util.Observer* interface and therefore the observer-pattern. It is designed to subscribe to data sources that implement the *java.util.Observable* interface. Log entries are sent to the *ObserverAdaptor* when the observed object calls its *notifyObersver* method, which results in a call to the *update* method of the *ObserverAdaptor*. The *update* method then creates an *Entry* of the corresponding type.

Condition As stated in section 3.2, one of the purposes of a *Condition* is to classify each *Entry* according to the threat level it poses. Any *Entry* generated by the *Anomaly Detection System* has already a classification assigned to it by the *AcceptedValue*, as described in section 4.3.2. This classification should be preserved in order to avoid having to evaluate it again using a very similar classification mechanism. Therefore the *Anomaly Detection System* provides an implementation of the *Condition* interface, the *AnomalyDetectionCondition*, that copies the classifications provided by the *AcceptedValue* from the log entry to the corresponding *Entry*. This is possible because the classification of the log entry is stored as a property of the *Entry*, which makes it accessible to the *AnomalyDetecionCondition*.

GUI The *Anomaly Detection System* does not have a GUI. The configuration is done, as in the *Forensic Analysis Functionality*, using the Spring Framework [13] and does not require a GUI. The output of the system is done using the *AnomalyDetectionLogger* and the *Forensic Analysis Functionality*, as described above. Therefore no GUI is required for the *Anomaly Detection System*.

The following sections will describe the details of the framework as well as the details of the reference implementation.

4.4.2 AcceptedValue

The *AcceptedValue* is a concept required to allow complex checks on the validity of one parameter value, as described in section 4.3.2. Each instance of an *AcceptedValue* is specific to a customer and a property of a certain class of *Entries*. The framework provides this as an abstract class with the following most important methods:

addNewValue This method is invoked by the *Rule* if no anomaly was detected, but a new value has to be added. This method is called when either a customer logs in for the first time, or an *Aspect* has changed within the accepted limits. It takes a single *Entry* as argument, from which the value to be added is extracted.

canAddValue This method evaluates if more values can be added. This might not be the case if the *AcceptedValue* contains a parameter value white list which may not be extended. It takes the *Entry* containing the corresponding value as parameter.

getPropertyName Returns the name of the *Entry* property that is checked using this *AcceptedValue*.

getViolationReturnCode Returns the classification of any contradiction detected in the *isAccepted* method.

isAccepted This is the main method of the *AcceptedValue*. It takes an *Entry* as parameter and evaluates if that *Entry* is accepted or not. It returns a boolean value indicating if a contradiction has been found.

reset Resets the *AcceptedValue* to its original state. This method allows the *AcceptedValue* to have temporary values, only consistent within one interaction with a customer. This can be used, for example, to create a *Aspect* of the browser, which may change over several interactions, but not within a single one.

storeAddedValue This method stores the new values of this interaction permanently by adding them to the accepted values. This could, for example, be the first geographic location from which the customer logs in.

The reference implementation has an implementation of the *AcceptedValue*, the *AcceptedValuesList*. This implementation allows the configuration of a list of accepted *java.lang.String* values. When the *isAccepted* method is called by the *Aspect* to perform a check on an *Entry*, the list is checked for a matching value. If no match can be found, the classification is stored and the *isAccepted* method returns *false*. The classification will be read during the evaluation performed by the *Rule*, as described in section 4.4.4.

The *AcceptedValuesList* offers the possibility to have three types of values:

Predefined Predefined values can be configured in the XML configuration file used to configure the *Anomaly Detection System*. The same values apply to all customers and they can not be changed by any interaction a customer has with the system. They do not need to be stored in the database since they do not change over time and can therefore be loaded from the configuration file. This type of value is used to set fixed boundaries to all customers of the system, for examples a white list of any parameter, such as a browser, or a black list of parameters, such as suspicious browser plug-ins.

Persistent Persistent values are the main value type used in the system because they are customer-specific. They are collected during a customer's interaction with the system and made persistent by storing them in the

database. They are designed to store the long-term behaviour of each customer, which is compared to the behaviour observed in the current interaction, which means that the stored values are compared to the corresponding ones of the current interaction. Examples of such values are the customer's PC, the browser and the geographical location.

Temporary Temporary values are customer-specific values collected during each interaction. They are not stored in the database as they are temporary. They are designed to track values that are not allowed to change during a single interaction, such as the exact browser version. This type of value is discarded when the interaction with the corresponding customer ends. This can be done by invoking the *reset* method of the *AcceptedValue*. The distinction between the temporary and the persistent value is made by applying the equate value to the *setStoreAddedValue* provided by the *AcceptedValue*.

Additional to the type of value, the *AcceptedValueList* allows the configuration of the amount of accepted values. This can be used to limit the number of accepted values of a parameter, for both temporary and persistent values.

The configuration required for a *AcceptedValue* is a name, a classification and a boolean value that specifies if the newly added values are stored permanently. The configuration of an *AcceptedValueList* requires two additional values, the number of values allowed and any predefined values.

4.4.3 Aspect

The *Aspect* is part of the framework and does not require any subtyping. It consist of a name, a unique identification and a key-value construct mapping a class of *Entries* to a list of *AcceptedValues*.

Its main functionality is implemented in the *applies* method, which takes a list of *Entries* and returns all information about any contradictions found and all information needed to add new values to any *AcceptedValue*. Additional values can be added to an *AcceptedValue* if a contradiction has been found and the configuration allows it. To do this, the *applies* method iterates over the list of *Entries* and loads the set of *AcceptedValues*, each of which is designed to check a single property of this specific *Entry* type. All *AcceptedValues* of this set are checked for contradictions. For each contradiction found, the information about the contradiction and, if adding a new value to the *AcceptedValue* is possible, the information required to do so is gathered. All this information is collected for the entire list of *Entries* and returned to the caller of the method, namely a *Rule*. The *Rule* can then call the *addValue* method if any new values need to be added.

The configuration of an *Aspect* requires only its name and the mapping from the class of the *Entries* to the corresponding *AcceptedValues*, as described above.

The *Aspect* and the *AcceptedValue* are not only the key concepts of the *Anomaly Detection System*, they are the only two objects represented in the database.

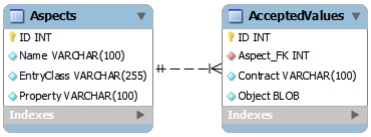

Figure 10: Database schema for the *Anomaly Detection System*

The two new tables required for the *Anomaly Detection System* are:

Aspects This table contains the properties of the *Aspect* as defined in the configuration file. Unlike the *Aspect* in the object model, its representation in the database is not specific to a customer but stores the data required to allow the assignment of the stored *AcceptedValue* to the corresponding *Aspect* of the object model. This is the name of the *Aspect*, the class of the *Entry* and the name of the property. The need for this is described in more detail in section 4.4.5.

Note: The content of this table is primarily stored in the configuration file. This makes the normalisation of the data contained in this table pointless. The configuration file forces the name of any bean to be unique. Therefore the name of the *Aspect* is unique and it does not require a representation in a separate table as demanded by the normalisation.

AcceptedValues The *AcceptedValue* table stores an object with the supertype *AcceptedValue*. Analogue to the storage of the *Entries* the whole object is serialized and written to the database. In addition to the object, the contract is stored in the table which allows the assignment of the *AcceptedValue* to the customer.

4.4.4 Rule

The *Rule* is an interface designed to combine several *Aspects* and determine if the contradictions found by these *Aspects* point towards an anomaly.

If an *Aspect* detects a contradiction, it includes all available information about it. This enables the implementation of the *Rule* interface to comprehend each contradiction and determine if the whole constellation of the contradictions point toward an anomaly or if the contradiction are caused by missing values.

In essence, this problem can be described as determining if the customer behaves differently than before or if the customer is in fact an impersonator with unauthorised access to the system. In the first case, the new values should be added to avoid having this problem in the future, and in the second case the interaction should be ended and appropriate actions should be taken. This makes the *Rule* the key element in detecting anomalies. The return value of the *evaluate* method indicates the minimum action that should be taken.

Beside the *evaluate* method described above, the interface defines the *reset* method. It is used to reset all temporary values stored in all *AcceptedValues*. This is done by invoking the *reset* method of all *Aspects*.

The *Anomaly Detection System* reference implementation provides a single implementation of the *Rule* interface, the *OneOrTheOtherRule*. It is a simple

implementation designed to allow a definable number of its *Aspects* to change. The idea is that the customer is allowed to change a subset of his behavioural aspects, but not all of them.

During the execution of the *evaluate* method of the *OneOrTheOtherRule* the number of *Aspects* that detected at least one contradiction are counted. If this number is smaller than the configured maximal number of changed *Aspects*, all new values that caused a contradiction are added to their respective *AcceptedValue* or otherwise an anomaly is reported. This is done by returning the highest classification of all contradictions found, which matches the action to be invoked by the system.

If an anomaly is reported, the *OneOrTheOtherRule* logs all contradictions. This is done using the *AnomalyDetectionLogger* which will create an *Entry* for every single contradiction. To avoid creating multiple log entries for the same cause, the configuration of the *OneOrTheOtherRule* offers the possibility to log a contradiction only once per interaction. If this is enabled, the *OneOrTheOtherRule* keeps track of the contradictions to only log them once.

The configuration required for each *Rule* will be as diverse as the *Rule* themselves, however, all of them will contain the list of *Aspects* to be checked. In addition to this list, the *OneOrTheOtherRule* requires the maximal number of *Aspects* that may detect a contradiction.

4.4.5 ComparisonBase

The *ComparisonBase* encapsulates the *Rules* and their *Aspects* for each customer. It is therefore customer-specific and only contains a list of *Rules*, each of which has a list of *Aspects*. This list of *Rules* is the only configuration required for the *ComparisonBase*.

The *ComparisonBase* does not offer any functionality on its own other than evaluating all its *Rules* and returning the highest classification returned by one of them. This functionality is implemented in the *performCheck* method.

Never the less, the *ComparisonBase* is very important. The whole *Anomaly Detection System* can be thought of as a stateful filter, where the state is changing over time and is specific to a single customer. The state itself is important to the system as a whole because it helps to increase the accuracy of the detecting mechanism.

The system is designed to change over time in terms of the configuration in place. This comes from the fact that the *Anomaly Detection System* is ideally run in collaboration with the *Forensic Analysis Functionality*. A system designed to, amongst others, identify new, not yet automatically detectable attacks to the system and help improve the security systems in place, such as the *Anomaly Detection System*. These not yet automatically detectable attacks should entail changes to the *Anomaly Detection System* to enable the system to detect them. Such a change in configuration should not result in the loss of the states gathered during the lifetime of the previous configuration. Therefore the new configuration has to be updated to the state of the previous one wherever possible. The functionality to do so is provided by the *ComparisonBase* and the corresponding *ComparisonBaseFactory*.

The *ComparisonBaseFactory* provides the *getComparisonBase* method that loads the current configuration and returns the *ComparisonBase* structure without a customer-specific state. The *ComparisonBase* has a *static loadByContract*

method that loads the entire *ComparisonBase* structure with its customer-specific state. This is done by loading a stateless *ComparisonBase* structure with the current configuration using the *ComparisonBaseFactory* method described above and all *AcceptedValues* associated with the contract. The part of the structure which is not based on a new configuration is updated to its last state, the rest remains uninitialised, as there is no data to initialise it with. This has to be done for each customer and the mechanism is invoked whenever a customer logs in.

Another issue handled by the *ComparisonBase* is storing the current state. This is handled by another *static* method called *store*. It takes a *ComparisonBase* and iterates over all *Aspects* of each *Rule*. To store the state of each *Aspect*, its *insertAcceptedValues* method, which stores all its *AcceptedValues*, is called.

4.4.6 AnomalyDetection

As described in section 4.3.2, the *AnomalyDetection* is the interface to the surrounding system. It encapsulates the entire functionality of the *Anomaly Detection System* by providing three methods:

getInstance The *AnomalyDetection* can only be instantiated once, because it manages a list of *ComparisonBases* each of which represents the current customer-specific state of the system. For each customer currently having an interaction with the system the specific state in form of the corresponding *ComparisonBase* has to be loaded.

Therefore the *getInstance* method is required since it returns the current instance of the *AnomalyDetection* or creates a new one if it doesn't exist yet.

Note: The *AnomalyDetection* has to be unique to avoid loading the *ComparisonBase* of the same customer twice. Therefore it has to be unique in the same scope as an interaction of a customer.

performAnomalyDetection This is the main interface method. It invokes the customer-specific anomaly detection mechanism and, depending on the return value of the *performCheck* method of the *ComparisonBase*, invokes an action to eliminate a threat to the system.

sessionClosed The number of *ComparisonBases* required to be loaded is equal to the number of customer interactions with the system. If an interaction is finished, the corresponding *ComparisonBase* has to be stored in the database for persistence and removed from the list of the *AnomalyDetection* to free up memory. Therefore each time a interaction has ended, the *sessionClosed* method is called which performs these two tasks.

4.4.7 Timing

The timing is a key element in detecting and successfully preventing an attack to the system, as described in section 4.1. Therefore the time required to detect an attack and react accordingly has to be analysed for each security system.

For the *Anomaly Detection System* to be considered as effective as the other security systems in place, it has to perform in a similar manner regarding the time needed to detect and prevent an attack. Therefore the performance of the *Anomaly Detection System* is compared to the *Request Validator* described in section 2.3.2.

The *Request Validator* is invoked synchronously each time a customer sends an *HTTP* request to the e-banking system. Therefore the detection of an attack will also be synchronously and performed within the duration of a request. For security reasons the average response time of an instance of the CLX.E-Banking system that is in use is unavailable and the average duration of the internal testing system have to suffice. The average duration of a request to the testing system is a little over three seconds (3271.03 milliseconds, calculated by dividing the sum of all durations by the number of request).

The *Anomaly Detection System*, in its current configuration and level of integration, is invoked each time the *Forensic Analysis Functionality* has performed its data collection, which is done once per second.

In addition to the delay caused by the invocation, the performance itself has to be analysed. As stated in section 3.2.2, the load to the *Forensic Analysis Functionality* and therefore the *Anomaly Detection System* is limited by the load to the entire e-banking system. This means that the performance of both the *Forensic Analysis Functionality* and the *Anomaly Detection System* will diminish less than the performance of the entire system. For that reason, the load to the system can be neglected in this context.

Since the reaction mechanism used by either system is the same one, its time consumption is irrelevant too. This allows the conclusion that the time required by the *Anomaly Detection System* to detect an attack is in the same order than the time required by the *Request Validator*.

5 Results

The goals of the thesis are derived from the description in section 1.2. As there are two systems described, the results are described accordingly.

The object of the *Forensic Analysis Functionality* was an extendable application. The result is a framework with a reference implementation. The framework provides the functionality required to run the system, such as the GUI, the mechanism to collect all data and the persistence functionality. This part of the system does not require any additional changes or configuration to be put to use. The reference implementation contains the actual implementations of the *DataAdaptors* with their corresponding *Entries* and the implementations of the *Condition*. This part of the system requires additional efforts to adapt the configuration to the individual instance and the environment of each CLX.E-Banking system installation. The implementation of additional *DataAdaptors* should not be necessary in the foreseeable future since a *DataAdaptor* for all currently available data sources was made during this thesis. Additional *Conditions* might be considered in order to be able to create more sophisticated filters.

The configuration made for this thesis is optimized for the internal testing installation of the CREALOGIX E-Banking AG where it has been subject to tests and performs as designed. *Entries* from all desired data sources are parsed and classified according to the *Conditions* in place. The set of provided *Conditions* allows stateless checks regarding any property of an *Entry*. The GUI is integrated into the administration interface of the CLX.E-Banking system and allows drawing a connection between different security related event. This is achieved by making all past and concurrent interactions with the system available in detail and offering the possibility to join them together to a *Case*. By doing so, the GUI allows the examination of all evaluation targets defined in section 1.2 and many more. The list of examination targets is ultimately limited by the data gathered by the set of active *DataAdaptors* in place. This allows the extension of the list of examinable evaluation targets by adding additional *Conditions*.

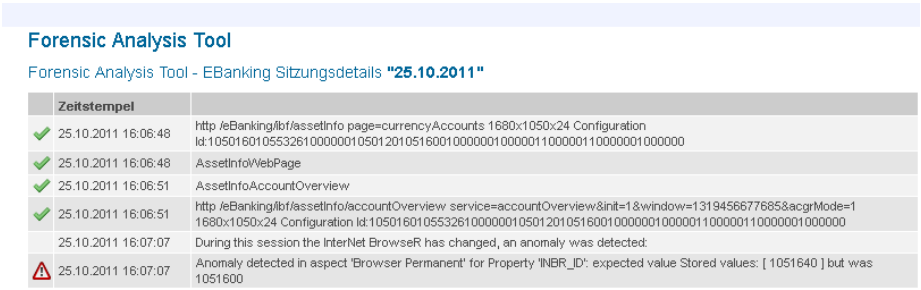

Figure 11: Screenshot of the detail view of an *EBankingSession* where an attack, specifically an anomaly, has been detected and the administrator has added a comment

The ability to check each interaction, regardless of the automatic detection of an attack, enables the administrator to comprehend attacks reported by either a

customer or a bank employee and not just the ones logged by a security system.

The CREALOGIX E-Banking AG knew about the necessity to have a security system analysing the customer behaviour over a longer period of time. A mechanism to log most of the required data was put in place long before the topics of this thesis were discussed. The actual implementation of the behavioural analysis was postponed. This thesis presented the potential to get the required background to design and implement this functionality.

The lack of research in this exact area introduces some uncertainty. This is accommodated for by implementing the *Anomaly Detection System* as a framework with a reference implementation.

The problem of identifying a behavioural aspect is not solved in the framework. It provides the concepts required to represent an behavioural aspect to the reference implementation along with the functionality to check each individual behavioural aspect and react according to the configuration of the reference implementation. In essence, the framework takes care of everything but the definition of the behavioural aspects and the classification of the differences detected in one.

Note: The definition of an behavioural aspect is done by providing the implementation and configuration of an *AcceptedValue*. The *Aspect* is the combination of all *AcceptedValues* which describe the same behavioural aspect, as explained in detail in section 4.4.2.

The reference implementation of this master thesis provides a relatively simple implementation of the behavioural aspect. It reduces each behavioural aspect to a list of accepted values that may differ for each customer, i.e. the *AcceptedValueList*. Using this, a set of aspects is defined, namely the PC, the browser and the geographical location. Differences detected are considered to be an anomaly if they originate from more than a single aspect. The reason this is considered to be relatively simple is the vast number of more accurate ways to describe a behavioural aspect than a list of accepted values.

Nevertheless, this concept proved to be effective in detecting *offline* attacks. For the system to mistakenly allow access to an imposter at least two of the three behavioural aspects have to be exactly emulated in this configuration. This proves to be very hard in practice, even if the required information is available.

In addition to detecting *offline* attacks, which the system is designed for, the *Anomaly Detection System* has shown remarkable accuracy in detecting session stealing attacks. The ability do to so comes from the fact that the attacker would have to access the stolen session, analogue to an imposter in a *offline* attack, emulating at least two of the three behavioural aspects perfectly.

Whether or not this thesis has any implications is ultimately depending on the CREALOGIX E-Banking AG and the banks purchasing the CLX.E-Banking system as the system will be put to use by them. However, both systems will, with the appropriate configuration, increase the system security considerably due to the newly acquired ability to detect and prevent *offline* attacks.

6 Conclusion

The conclusion section recaps the goals and the contributions of the thesis and suggests future work.

6.1 Thesis Contribution

As stated in section 2, the goal of the thesis was to increase the security of the CLX.E-Banking system. This is achieved by both systems individually or, even more so, by both systems combined.

The *Forensic Analysis Functionality* allows the surveillance of the current security systems and the state of the system as a whole. It enables the administrator to track any interaction with the system, in particular an attack and the responses of each security system can be tracked. By collecting all information about similar attacks, the administrator can provide the background for improvements in current security systems or the cause to create entirely new ones. The big benefit this implementation provides is the fact that the system can be viewed not just in terms of the different components that make up the system but in terms of the customer interaction. It allows the administrator to examine the interactions and therefore comprehend them in any regard.

The *Anomaly Detection System* as it is implemented in this thesis is very effective in detecting attacks using the relatively simple configurations described in section 5 and 4.3.1. As stated in section 5, CREALOGIX E-Banking AG knew about the need for a system like the *Anomaly Detection System*. This implementation confirms this. Beside the ability to detect session stealing attacks, it is the only security system of the CLX.E-Banking designed to detect *offline* attacks and succeeds in doing so. The effectiveness of the system using the simple configuration described in section 5 is remarkable. It is even more so, if the complexity of the configuration is compared to the possibilities offered by the system. Therefore is the performance of the system the best recommendation for its usage.

6.2 Future Work

Forensic Analysis Functionality This system works well and can be put to use instantly. While the look and feel of the administration interface was adapted during implementation, it requires additional changes to comply the corporate design and identity of CREALOGIX E-Banking AG.

The configuration created for the reference implementation should be revisited to make sure all data not required by the system itself or the *Anomaly Detection System* is discarded. This is a task to be done each time the configuration of either system changes.

To guarantee the long term system stability the *Forensic Analysis functionality* and the data it stores in the database should be subject to the system maintenance tools. This is currently not the case.

Anomaly Detection System The ability to detect an intrusion into an e-banking system using behavioural analysis seems to be very convenient,

however, there is a lack of research in this area. The representation for a behavioural aspect chosen in the reference implementation is reasonable and useful, but not founded on solid research. To increase the security provided by this system, the exploration of more efficient and more accurate representations is therefore highly recommended. Examples of ideas to enhance the system in this way are a stateful representation taking into account the browsing behaviour of the customer and a representation determining the validity by the use of a formula or regular expression.

Extensions and new configurations of this system could profit greatly from the results of research in the areas of intrusion detection in applications and analysis of browsing behaviour on web application. In addition, attacks either reported or automatically detected should be examined and the system adapted accordingly.

7 Glossary

CSRF Cross - Site Request Forgery [3]

EJB Enterprise Java Bean

GUI Graphical User Interface, interface that allows the user to interact with the application

fat client Client side program of a client server application that provides additional functionality independent of the server

HTML HyperText Markup Language, markup language for web pages

HTTP Hypertext Transfer Protocol

JDBC Java Database Connectivity, API to access a database from Java

Java Object - oriented programming language

OS Operating System

SID Session Identification, piece of data that identifies a session between client and server

SQL Structured Query Language, programming language to manage relational data

SSL Secure Sockets Layer, protocol which provides secure communication over the internet

URL Uniform Resource Locator, character string that specifies the location of a resource

WAF Web Application Firewall, filters internet traffic (HTTP) between client and server

XHTML eXtensible HyperText Markup Language, an extended version of HTML

XML Extensible Markup Language, is a set human and machine readable data format

XSLT eXtensible Stylesheet Language Transformations, an XML based language to transform XML documents

XSS Cross - Site Scripting [2]

References

[1] E-Banking solution developed by CREALOGIX E-Banking AG, *http://www.crealogix.com/leistungsangebot/e-banking/produkte/clxe-banking.html*, 2011 (accessed December 5, 2011).

[2] "CERT Advisory CA-2000-02 - Malicious HTML Tags Embedded in Client Web Requests", CERT, February 2nd, 2000

[3] Jesse Burns, 2005, Information Security Partners, LLC. "Cross Site Reference Forgery An introduction to a common web application weakness".

[4] The Largest Swiss Banks by Total Assets, *http://moneystockstycoons.com/bank-lists/switzerland-banks/*, 2011 (accessed December 5, 2011).

[5] All of the 25 largest bankes have a web based e-banking solution: This can be assessed visiting the websites of the top 25 banks listed in [4].

[6] A. Hiltgen, T. Kramp, T. Weigold, *Secure Internet Banking Authentication*, Security and Privacy, IEEE, March 2006.

[7] Paes de Barros, Augusto (15 September, 2005). "O futuro dos backdoors - o pior dos mundos". Sao Paulo, Brazil: Congresso Nacional de Auditoria de Sistemas, Segurana da Informao e Governana - CNASI. Retrieved 2009-06-12.

[8] Felix, Jerry and Hauck, Chris (September 1987). "System Security: A Hacker's Perspective". 1987 Interex Proceedings 1: 6.

[9] Lance James, *Phishing exposed*, 1 edition , Syngress Publishing Inc.(January 20, 2006), Pages 2, 31.

[10] Fernando de la Cuadra, (07 March, 2005), *http://www.crime-research.org/news/03.07.2005/1015/*, 2011 (accessed December 5, 2011).

[11] Justin Clarke, *SQL injection attacks and defense*, Syngress Publishing Inc. 2009.

[12] Oracle WebLogic Server, *http://www.oracle.com/technetwork/middleware/weblogic/overview/index.html*, 2011 (accessed December 5, 2011).

[13] Spring Framework, *http://www.springsource.org/*, 2011 (accessed December 5, 2011).

[14] A. Sharma, Z. Kalbarczyk, R. Iyer, J. Barlow, *Analysis of Credential Stealing Attacks in an Open Network Environment*, Network and System Security (NSS), 2010 4th International Conferece, 15. November 2010.