Automated Modelling of Security Incidents to represent Logging Requirements in Software Systems

Fanny Rivera-Ortiz

University College Dublin & Lero
Dublin, Ireland
fanny.riveraortiz@ucdconnect.ie

Liliana Pasquale

University College Dublin & Lero
Dublin, Ireland
liliana.pasquale@ucd.ie

Abstract

In 2017 the Open Web Application Security Project (OWASP) has identified insufficient logging and monitoring as one of the top ten security risks. Attackers can exploit insufficient logging in software systems to cause harm to an organisation while being undetected for long periods of time. Therefore, software systems used within an organisation should perform logging to collect data relevant to detect and/or diagnose potential security incidents. However, when implementing logging functionalities, software developers either do not log enough information or log too much information. In this paper, we provide an approach to help developers decide where to log and what to log for security purposes. Our approach allows a security engineer to replay potential security incidents on an instrumented version of the software system and generate automatically a model of such incidents. These are represented as a UML sequence diagram that contains the relevant method invocations occurring during and incident, without providing a representation of the entire software behaviour. Because our model refers to concrete system components, it provides immediate guidance to developers about what methods execution should be logged for security purposes.

CCS Concepts

• **Security and privacy** → **Software and application security**;
Software reverse engineering;

Keywords

security, incident, software, systems, logging, logs, modelling, representation, threat modelling

ACM Reference Format:

Fanny Rivera-Ortiz and Liliana Pasquale. 2020. Automated Modelling of Security Incidents to represent Logging Requirements in Software Systems. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020)*, August 25–28, 2020, Virtual Event, Ireland. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3407023.3407081>

1 Introduction

Today, the software is eating the world [1]. Almost every organisation worldwide uses software to perform its business processes

and deliver its services [2]. However, offenders can exploit software vulnerabilities to perform malicious activities against organisations and society. During 2020, because of the global coronavirus pandemic, the FBI has reported an increase of 400% on the daily security incidents reports received [3]. Thus, it has become more important than ever for organisations to be ready to detect and diagnose potential security incidents.

The Open Web Application Security Project (OWASP) provides resources to organisations to develop secure software systems. One of the top ten security risks identified by OWASP is “Insufficient logging and monitoring”. This security risk allows criminals to further attack systems, maintain persistence, pivot to more systems, and tamper, extract or destroy data [4]. Attackers can exploit insufficient logging to cause harm to an organisation while being undetected for long periods of time. For example, in November 2018, Marriott International announced to be targeted by a massive data breach—the second largest in history [5]—where approximately 327 million of their customers were affected and their confidential information was exposed. An investigation conducted revealed that attackers were able to access customers’ confidential information since 2014 [6]. This means that offenders were undetected for about 4 years [7]. Implementing logging in Marriott’s reservation systems could have helped to discover security incidents more promptly.

Logs can provide information useful during a digital investigation to identify who, what, when, where, and how a security breach occurred [8, 9] and also to support non-repudiation of threats [9]. However, when implementing logging functionalities, software developers either do not log enough information or they log too much information [8, 10]. Logging too little poses the risk to miss relevant information (e.g. username, timestamp, description, hostname, and/or IP address) for posterior analysis. Logging too much [10] has the drawback of adding more code to write and maintain, producing non-informative logs having limited usefulness, and increasing usage of resources (e.g., CPU and I/O). Although software systems can easily store trillions of log messages, the security personnel can barely analyse and respond to a dozen of issues per day [8]. Therefore, developers should implement logging functionalities in software systems that ensure that only key information [10] relevant to security incidents is collected. Logs generated by a software system can provide digital evidence [11] that can explain how an incident occurred [12] and can endure legal scrutiny in a court of law [13, 14]. A system able to produce such evidence is so-called *forensic-ready* [15].

The purpose of our research is to help software developers to build forensic-ready software systems, which can implement logging functionality to detect and diagnose potential security incidents. To achieve this aim, developers need to detect *where to log*,

This work is licensed under a Creative Commons Attribution International 4.0 License.
ARES 2020, August 25–28, 2020, Virtual Event, Ireland
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8833-7/20/08.
<https://doi.org/10.1145/3407023.3407081>

which refers to the locations where logging statements should be placed, and *what to log*, which refers to the information recorded in the log [10, 16]. In our previous work [16] we presented some initial ideas about how to automate the implementation of logging functionalities in software systems. Our approach, called Forensic-Ready Logger, aims to elicit a set of potential software misuse scenarios, expressed as annotated sequence diagrams. These diagrams can then be used to identify the exact location where logging instructions should be placed and the information they should log. In this paper, we focus on the elicitation and the automated representation of software misuse scenarios. These scenarios represent concretely how security incidents can happen in software systems. In other words, they identify the sequence of relevant software methods that are invoked during an incident and the software components that are affected. Such representation can allow developers to determine where logging should be performed (e.g., when a method is executed) and the information that should be logged (e.g., invoked method and the value of its input/output parameters).

We require a security engineer to replay potential security incidents on an instrumented version of the software system, which generates automatically a model of such incidents. These are represented as a UML sequence diagram that contains the relevant method invocations occurring during an incident, without providing a representation of the entire software behaviour. The sequence diagram can be subsequently annotated to allow a security engineer to identify additional information that should be logged and/or conditions determining whether logging should be performed. We also describe the prototype we implemented to support our approach. We used our prototype to automatically generate the sequence diagram representing a security incident in a human resources management software system.

This paper is organised as follows. In Section 2, we provide some background information about logging and present existing related work. In Section 3, we present a running example to motivate our approach. In Section 4, we provide an overview of our approach. In Sections 5 and 6, we describe the activities that, respectively, a software engineer and a security engineer should perform to use our approach. Finally, in Section 7, we conclude the paper and discuss future research directions.

2 Background and Related Work

This section presents some background information about logging and gives an overview of previous work aimed to represent security incidents for different purposes.

Background. A security incident refers to malicious activities that can violate either an organization policy or a law [16, 17]. A log is a file generated by a software system that contains a set of events, i.e. changes, that happened in such system [8], for instance, the occurrence of an error, the creation of new information entities. Logs provide evidence to determine what happened after a security incident occurred [8, 9, 16] in a software system. For example, the analysis of the timestamp and the event description of a log may help to determine, respectively when an incident happened and what happened.

To implement logging statements in software systems, developers can use libraries, such as Log4j [18] for Java-based software systems.

However, developers often focus on functional and performance requirements [19] or fixing bugs in software systems [20] and are not always familiar with potential security incidents that can occur in software systems [16, 19]. Hence, developers may not have the expertise to implement logging for security purposes [21], because they may not be aware of how the software can be misused [19, 20] and what information should be collected to detect potential software misuses [16].

Related Work. Different modelling approaches have been proposed in previous work to represent security incidents in software systems. Misuse cases [22] describe undesired software system behaviour from a security perspective. Misuse cases are represented as UML Use Cases diagrams which are linked to security requirements in software systems. However, they are not linked to the software internal components such as classes, methods, and attributes. Thus, misuse cases cannot be used directly to identify where and what to log to detect and diagnose security incidents.

Abuse Frames [23] are diagrams that represent how a system in a specified domain can be threatened to identify security vulnerabilities. An abuse frame represents anti-requirements (i.e. the requirements of malicious attackers), the assets within the domain under attack, the interface between the attacker, the assets that can be used to achieve the anti-requirement, and the domain that is imposing the threat. Abuse frames allow identifying vulnerabilities but have the limitation of representing incidents as a single step action. Thus, they cannot be used to identify the events of an incident that require to be logged.

The Security Domain Specific Visual Language (SecDSVL) [24] allows modelling security objectives, security requirements, enterprise security architecture, and security controls operated. This is used in combination with a system model that captures the system features, architecture, and main components such as classes, and deployment details. Both models are merged by security administrators to generate a unique enterprise security model that helps the management of the security process within the organisation. However, this approach is semi-automated and still relies on a human operator to merge the security and the system model. This can expose the approach to errors and omissions.

Extended Data Flow Diagrams (EDFDs) [25] are traditional data flow diagrams but are annotated to provide additional information about security threats. The EDFDs can be analysed automatically using security rules defined in a knowledge base. The analysis leads to a list of tackled security problems and a list of security flaws that have not been yet mitigated. However, this approach requires creating the EDFDs manually. This can be prone to errors considering that a software system can have a multitude of methods, which are executed in a different order at runtime.

The Meta Attack Language (MAL) [26] allows generating attack graphs automatically for a specific software system domain (cloud computing or embedded systems). Attack graphs represent the dependencies between the steps that may be performed by the attacker to compromise the software system. MAL also produces an estimate of the time necessary for the attacker to compromise the assets of interest. However, every element that is part of the original attack graph and the application domain needs to be represented manually. Also, a generated attack graph does not ensure traceability to the underlying software system components.

Previous work [22–26] allows generating threat models and providing assistance to companies to manage the security process. However, it is not suitable to represent security incidents in software systems and identify method executions that should be logged, automatically.

3 Motivating Example

This section presents a Human Resources Management software system (HRM) example [16]¹, which we use to motivate and showcase our approach.

This system supports the submission and approval of travel requests. It includes three main modules. 1) Employees: it allows adding, modifying, deleting, and viewing employees; 2) Users: it allows adding, modifying, deleting, and viewing users of the HRM software system and their credentials; 3) Travel Requests: it allows creating, modifying, approving, deleting, and viewing travel requests. Furthermore, the HRM software system has the following user-levels or user-privileges: a) Admin: can access all the modules of the HRM software system; b) Manager: can only access modules Employees and Travel Requests; c) Employee: can only access module Travel Requests, and cannot approve travel requests.

An insider, Henry Smith, who is the System Administrator of the HRM system, is going on a business trip. Thus, he creates a travel request while his line manager, Arthur Jones, is away on holiday. Henry uses his privileges as a System Administrator to modify Arthur’s credentials and impersonate Arthur, to approve his travel request.

Figure 1 describes the sequence of actions characterizing this security incident.

- 1) Henry connects to the HRM software system with the user-level: Administrator.
- 2) Henry accesses the Travel Request module and creates a new Travel Request.
- 3) The Travel Request is created and sent to Arthur. However, Arthur is on holiday and cannot approve of it.
- 4) Henry needs approval for his travel request and connects as Administrator to the HRM software system.
- 5) Henry goes to the Users module and modifies Arthur’s password and exits the HRM software system.
- 6) Later, Henry misuses his privileges as a System Administrator because he impersonates Arthur, by connecting to the HRM software system using the modified Arthur’s credentials. The HRM software system displays a message: “A Manager has connected into the HRM software system.”
- 7) Finally, Henry accesses the Travel Request Module, reviews the travel request and approves it.

This scenario represents a security incident because it violates one of the organization’s policies, i.e. “*only a manager can authorise a travel request*”. If adequate logging is not performed, this security incident can go undetected or the investigation about this incident may be difficult because relevant evidence that indicates the offenders’ identity and the incident actions are missing. Note that it is not possible to prevent Henry from modifying the users’ credentials, because he requires this access privilege to perform his duties as a

Figure 1: Incident example representing a misuse of the HRM software system.

System Administrator. Thus, it is necessary that the HRM system to record user actions that prove software misuses and identifies victims and offenders [27].

4 Automated Modelling of Security Incidents

Our approach is aimed to generate a model of security incidents automatically. This model represents the steps of security incidents by referring directly to concrete software system components (e.g., packages, classes, and methods). From now on we will refer to our approach as Forensic-Ready Logger (FRL).

To generate a model of security incidents, our approach involves two participants: a) a software engineer, who has the technical knowledge about the software system, and b) a security engineer, who has the security knowledge about potential incidents in the software system. The security engineer uses an instrumented version of the original software system to simulate the activities of an offender. While the software engineer is involved in the instrumentation of the original software system. This is necessary to generate a model of potential security incidents automatically.

As shown in Figure 2, the software engineer instruments the original software system with the aspects necessary to generate the incident model (step 1). Then, the security engineer replays the

Figure 2: Activities performed by the Security Engineer.

actions of an incident (step 2). For example, s/he can replay all the steps of the incident example described in Figure 1. Afterwards, the instrumented software system generates a UML sequence diagram representing the incident (step 3).

The components of the sequence diagram are software system classes, while the messages exchanged between components represent method invocations. Message exchanges are displayed in the

¹A Java-based implementation of the system is available publicly at <https://github.com/friveraortiz/Projects.git>

order in which their corresponding methods are executed when an incident is replayed by the security engineer. The generated incident model allows a software engineer to quickly identify the method invocations that should be logged, and also, the input/output parameters that should be recorded.

We assume that the software system is instrumented using Aspect Oriented Programming (AOP) [28]. This is a programming paradigm that allows modularising crosscutting concerns without modifying the implementation of the software system. An aspect encapsulates a pointcut, which selects specific execution points in the software system, called join points. An aspect also includes an advice, which represents the functionality executed when one of the join point is reached. In this work, an aspect should intercept the invocation of methods that are executed during a security incident. The advice should implement the functionality necessary to represent the method invocation as a message exchange in the UML sequence diagram representing the incident. The task of the software engineer is to provide information necessary to identify the methods whose execution should be represented in the model of the security incident. These are the methods that are relevant for security logging purposes.

We assume that security incidents result in access to and/or modification of valuable information in the database. This decision relates to the fact that one of the most severe technical impact identified in the OWASP Top 10 vulnerabilities [4] is related to access and/or modification of sensitive information. Additionally, King and Williams [9, 29] proposed as one of their principles to create logging mechanisms in software systems to log all data transactions. Chuvakin et al. [8, 30] also advise logging events in software systems related to data changes (e.g., creation, modification, consultation, and destruction).

As shown in Figure 3, to instrument the original software system, the software engineer provides information about the software system such as the methods that begin, end, and connect to the application. Also, information about the source code language, and the DBMS (step 1).

Figure 3: Activities performed by the Software Engineer.

Then, our approach can automatically identify all the methods that directly or indirectly modify valuable information in the database (step 2). This allows selecting only those methods (in the *Method List*) that can potentially be relevant to security incidents ruling out irrelevant ones. Later, our approach creates two aspects that are tasked to intercept the execution of the methods in the Method List and to represent them as message exchanges in the sequence diagram representing the incident (step 3).

In the next sections, we will describe in more detail the activities performed by the software and the security engineer (respectively, in Section 5 and Section 6).

5 Software Engineer Activities.

We implemented our Forensic-Ready Logger (FRL) approach² assuming that the target software system is implemented in Java and uses MySQL as a DBMS. The FRL utilizes two folders: a) *Input*, where the information about the original software system is stored, and b) *Output*, where the Forensic-Ready Logger generates the aspect file necessary to instrument the target software system.

As shown in Figure 3, the software engineer *Provides system information* (step 1) by placing in the Input Folder the following resources and information:

- 1) The source code of the software system to be instrumented.
- 2) A JAR file of the software to be instrumented.
- 3) A configuration file (*ftrlconfig.properties*) including additional information about the software system, such as the language in which it is implemented, the DBMS, the methods that start and terminate the application, and those that are invoked by classes representing graphical interface elements.

For example, in the HRM software system, the software engineer provides the following information in the *ftrlconfig.properties* file: a) Method that starts the application: *gui.HRMApp.main*, b) Method that terminates the application: *void exit*, c) Method that allows users to login to the application *gui.LoginFrame.new*, d) the Programming language: *Java*, and e) DBMS: *MySQL*.

Once the software engineer provided all the system information in the *ftrlconfig.properties* file, the Forensic-Ready Logger can *Generate* (step 2) a list of all the method invocations from the original software system that can potentially be represented as a message exchanges in the incident model.

A software system can have a multitude of methods. Generating a UML sequence diagram that represents all the method invocations occurred during a security incident, might result in an extremely extensive UML sequence diagram because it might involve the entire software behaviour. Such behaviour might include the graphical user interface and the business logic of the software system. Such an extensive UML sequence diagram is *unmanageable* and might make it difficult for a security engineer to understand the locations (e.g., where to log) and the places (e.g., what to log) to detect the security incident. To reduce the number of methods that will be included in the UML sequence diagram our approach only represents the methods that are invoked when the misuse scenario is replayed in the software system. Also, we avoid generating any methods invoked from graphical interface events (e.g., cursor movements, menus, buttons, text fields). Finally, we ensure that only methods that result in data access or changes in the database are represented in the incident model [8, 9, 29, 30].

To identify methods that result in data access and/or modification in the database the Forensic-Ready Logger performs the following steps: 1) The FRL reads the names of the classes and methods utilized to perform data access/change in the DBMS. These are originally stored in the *database_operation* table and vary depending on the

²The source code of our Forensic-Ready Logger approach is available publicly at <https://github.com/friveraortiz/Projects.git>

language in which the software system is implemented and the DBMS. 2) The FRL checks whether any of the software system methods match any of the methods included in the *database_operation* table. If so, the method will be *included* in the incident model.

For example, in the HRM software system, there is the method: *model.TravelRequestDatabase.save*, where *model* is the package name, *TravelRequestDatabase* is the class name and *save* is the method name. The HRM software system uses Java as source code language. The *database_operation* table contains the classes and methods used in Java to perform data changes according to Java Documentation [31]. Then, the FRL validates if the *save* method contains one or more values stored on the *database_operation* table. In this case, this condition is true because the *save* method contains a variable called *insertStatement* declared as *PreparedStatement* type, which invokes the *executeUpdate* method. The *PreparedStatement* is a class used in Java to perform data changes and the *executeUpdate* is a method used in Java to perform data changes. Therefore, the Forensic-Ready Logger determines that the *save* method performs data changes and will be *included* in the incident model.

The Forensic-Ready Logger makes the following steps to validate if a method creates Graphical User Interface (GUI) elements in the software system: 1) The graphical_user_interface table, which is present in the FRL database, stores the values of the libraries or frameworks used in a source code language to create graphical user interface elements. Then, the FRL loads the information stored in this *graphical_user_interface* table. 2) For every method in the software system, the FRL validates if the method contains one or more values stored in the *graphical_user_interface* table. If this condition is met, the method will be *excluded* from the incident model.

For example, in the HRM software system, there is a method called *gui.LoginFormPanel.layoutComponents*, where *gui* is the package name, *LoginFormPanel* is the class name and *layoutComponents* is the method name. The *graphical_user_interface* table contains the libraries used in Java to create graphical user interfaces according to the Stack Overflow site [32]. Then, the FRL validates if the *layoutComponents* method contains one or more values stored on the *graphical_user_interface* table. In this case, it is true, because the *graphical_user_interface* method is declared as part of the *LoginFormPanel* class. Such class receives inheritance from the *JPanel* class (e.g., public class *LoginFormPanel* extends *JPanel*). The *Jpanel* class is declared in *javax.swing*, which is a framework used in Java to create graphical user interface elements. Therefore, the Forensic-Ready Logger determines that the *layoutComponents* method will be *excluded* in the incident model.

Once we reviewed the criteria that the Forensic-Ready Logger uses to decide which methods invocations will be present in the incident model, the software engineer performs the next step according to Figure 3: *Creates* (step 2). In this step, the software engineer runs the Forensic-Ready Logger. As an outcome, two lists are generated:

1) *Methods that perform data changes*: This list shows all the methods that carry out in the software system: a) data transactions *directly* towards a database (e.g., creation, modification, consultation, and destruction), and b) do not create graphical user interface elements (e.g., screens, menus, buttons, text fields).

Figure 4: Some methods that perform data changes in the HRM software system.

For example, Figure 4, shows an excerpt of the method list generated by the FRL that perform data changes in the HRM software system. On the left side, there are blue circles with numbers, which indicate the order of invocation of such methods in the HRM software system. Furthermore, the FRL provides the full name of the method, which consists of three elements: package name, class name, and method name. In number 3, the full method name is *model.TravelRequestDatabase.save*. This consists of three elements: a) package name: *model*, b) class name: *TravelRequestDatabase*, and c) method name: *save*.

2) *Call-hierarchy of the methods that perform data changes*: This list presents the *calling order* of the methods that directly perform data changes and the methods that invoke such methods, which also do not create graphical user interface elements in the software system.

For example, Figure 5, shows an excerpt of the call-hierarchy method list generated by the FRL that perform data changes in the HRM software system.

Figure 5: Some methods in the call-hierarchy list that performs data changes in the HRM software system.

The blue circles with numbers indicate the order of invocation of the methods and the hierarchy or level. The first method: 1) *model.TravelRequestDatabase.save* is the method that directly perform data changes in the HRM software system. The second method: 2) *controller.TravelRequestController.save* calls the method a), but do not create graphical user interface elements in the HRM software system. The third method: 3) *test.TestTravelRequestDb.main* calls

the method *a*) and also do not create graphical user interface elements in the HRM software system..

The *call-hierarchy* list is the definitive source for the Forensic-Ready Logger to include all these methods in the incident model. This list narrows down considerably the number of methods of the software system because every method in this list meets two conditions: *a*) perform data changes, and *b*) does not create graphical interface elements.

The software engineer continues with the next step according to Figure 3: *Creates* (step 3). This step refers to the generation of the aspect files using the Forensic-Ready Logger.

Then, the software engineer runs the Forensic-Ready Logger. As outcome two aspect files are generated: *a*) *FrlDatabaseMethods.aj*, and *b*) *FrlStart.aj*. These two aspect files are personalized to be used in the HRM software system.

a) *FrlDatabaseMethods* Aspect File.

This is an excerpt of the *FrlDatabaseMethods.aj* aspect file:

```
01: public aspect FrlDatabaseMethods
02: {
03:   pointcut databaseMethods():
04:     call(boolean model.UserDatabase.validateUser(..)) ||
05:     call(boolean controller.UserController.validateUser(..)) ||
06:     call(model.UserLevel model.UserDatabase.getUserLevel(..)) ||
07:     call(void model.TravelRequestDatabase.save(..)) ||
08:     call(void controller.TravelRequestController.save(..)) ||
09:     before(): databaseMethods()
10:   {
11:     callerClassObj1 = thisEnclosingJoinPointStaticPart.
12:       getSignature().getDeclaringType();
13:     getCallerClassDetails(callerClassObj1);
14:     calleeClassObj1 = thisJoinPoint.getTarget().getClass();
15:     getCalleeClassDetails(calleeClassObj1);
16:     method = thisJoinPoint.getSignature();
17:     nameMethod = method.getName();
18:     methodArgs = thisJoinPoint.getArgs();
19:     fullMethod = nameMethod + "(" + ;
20:       Arrays.toString(methodArgs) + ")";
21:     note = currentUser + colorNote + callerClass ;
22:     updateSeqDiagTextFile(callerClass, calleeClass,
23:       fullMethod, note, callerClassLine, calleeClassLine);
24:   }
25:   pointcut connect():
26:     execution(gui.LoginFrame.new(..));
27:     before(): connect()
28:   {
29:     line = startDivision + projectName + endDivision;
30:     currentUser = userName;
31:   }
32: }
```

The aspect file: *FrlDatabaseMethods* includes two pointcuts.

1.- *databaseMethods*: generates the components of the UML sequence diagram that will be added while the security engineer is replaying the misuse scenario in the HRM software system. Such components are: *callerClass*, *calleeClass*, *fullMethod*, and *note* (lines: 22 and 23). The *databaseMethods* pointcut (line 03) starts with the joint points (lines 04-08). When these methods are invoked in the

HRM software system will execute the code specified in the advice (line: 09). Such methods (lines 04-08) were generated by the Forensic-Ready Logger as part of the *call-hierarchy method* list.

The *databaseMethods* advice (lines: 09-24) include the code that generates the components of the UML sequence diagram:

- a) *caller Class*: sends the message to the receiver component (lines 11 -13). It can be either an actor, which is a human user of the HRM software system (e.g., Henry Smith) or a participant which is a system element (e.g., controller.UserController class).
- b) *callee Class*: receives the message from the sender component (lines 14 -15). It can be only a system component (e.g., model.UserDatabase class).
- c) *full Method*: is the message exchanged between the sender and the receiver components (lines 16 - 20).
- d) *note*: is a rectangle in an aqua colour that presents the sender component that creates graphical user interface elements (e.g., screens, menus, buttons, text fields) in the HRM software system (line 21).

2.- *connect*: generates a division in the UML sequence diagram that indicates a user has connected to the HRM software system. The *connect* pointcut (line 25) starts with a joint point (line 26). This joint point includes one method that when is invoked in the HRM software system, will execute the code specified in the advice (lines: 27-31). The *connect* advice creates a white rectangle that contains the text: "Connects to the HRM Application". This message indicates when a user login into the HRM software system.

b) *FrlStart* Aspect File.

This is an excerpt of the *FrlStart.aj* aspect file:

```
01: public aspect FrlStart
02: {
03:   pointcut start():
04:     execution(void gui.HRMApp.main(..));
05:     after(): start()
06:   {
07:     createSeqDiagramTextFile(umlSeqDiagTextFile);
08:   }
09:   pointcut end():
10:     call(void exit(..));
11:     before(): end()
12:   {
13:     updateFiles();
14:   }
15: }
```

The aspect file: *FrlStart* contains two pointcuts:

- 1) *start*: creates a file called: *IncidentSequenceDiagram.txt*. This file stores the components of the UML sequence diagram, which will be added while the security engineer is replaying the misuse scenario in the HRM software system. It has a unique joint point (line 04), which when invoked will execute the code indicated in the advice (lines 05-08). The advice (lines 05-08) creates the initial version of the *IncidentSequenceDiagram.txt* text file as an *empty* file.
- 2) *end*: generates the final version of the text file: *IncidentSequenceDiagram.txt*. This file contains all the methods invocations that occur while the security engineer is reproducing the misuse scenario in the HRM software system.

This pointcut also generates an image file named: *IncidentSequenceDiagram.png*. This file contains the graphical representation of the UML sequence diagram. It has a unique joint point (line 10), which when is invoked will execute the code indicated in the advice (lines 11-14). The advice creates the *IncidentSequenceDiagram.txt* and *IncidentSequenceDiagram.png* files.

6 Security Engineer Activities.

After the aspect files are generated, the security engineer is ready to participate in our Forensic-Ready Logger approach as shown in Figure 2.

The software engineer *Loads the aspect files* (step 1). This step includes *a)* copy the aspect files in the folder where is located the software system, and *b)* create a package named *fri* in the software system to load such aspect files. Then, the security engineer performs the next step: *Reproduces* (step 2). In this step, the security engineer replays all the activities that led the misuse scenario to occur in the software system. For example, in the HRM software system, the security engineer carries out all the activities described in our incident example (Figure 1). Later, the software system is instrumented with the aspect files and performs the final step: *Creates* (step 3). In this step, the software system generates the incident model. Such a model is a UML sequence diagram that contains all the methods that are invoked while the security engineer is reproducing the misuse scenario in the software system.

Figure 6 shows an excerpt of the Incident Model generated when the Admin: Henry Smith creates a new travel request in the HRM software system.

Figure 6: Incident Model for the Admin: Henry Smith in the HRM software system.

This UML sequence diagram contains is described as follows. 1) Connects is a message that indicates when a user logs into the HRM software system. It is represented by a white rectangle. 2) caller Class is the component that sends the message. It is represented by the actor (e.g., User_1) and the note, which is an aqua rectangle. 3) callee Class is the component that receives the message. It is represented by the navy blue rectangle. 4) full Method is the method invoked in the HRM software system. It is represented by the arrow.

On Figure 6, starting from the left side, there is the User_1, which represents: *Henry Smith*. Then, the *gui.LoginFrame* component (caller Class) invokes the method: *validateUser* (Number 1) towards to the *controller.UserController* component (callee Class). Later, the method: *validateUser* (Number 2) is invoked. Both methods (Numbers 1 and 2) are related to grant access to *Henry Smith*

to connect to the HRM software system. Soon after, the method: *getUserLevel* (Number 3) is invoked. This method is when *Henry Smith* is authorized with user-level as Admin in the HRM software system. Finally, the methods with numbers 4 and 5 are invoked. Such methods are related to the creation of a new travel request in the HRM software system.

The Figure 7 shows an excerpt of the Incident Model generated when the Manager: “Fake Arthur Jones” authorizes the existing travel request in the HRM software system.

Figure 7: Incident Model for Manager: Fake Arthur Jones in the HRM software system.

On Figure 7, starting from the left side, there is the User_1, which represents: “*Fake Arthur Jones*”. Then, the *gui.LoginFrame* component (caller Class) invokes the method: *validateUser* (Number 1) towards to the *controller.UserController* component (callee Class). Later, the method: *validateUser* (Number 2) is invoked. Both methods (Numbers 1 and 2) are related to grant access to “*Fake Arthur Jones*” to connect to the HRM software system. Soon after, the method: *getUserLevel* (Number 3) is invoked. This method is when “*Fake Arthur Jones*” is authorized with user-level as Manager in the HRM software system. Finally, the methods with numbers 4 and 5 are invoked. Such methods are related to the authorization of the existing travel request in the HRM software system. This is the exact moment when the misuse scenario occurs in the HRM software system. This UML sequence diagram helps the security engineer to visualize this. Later, it will be necessary to add logs statements at this point to detect this security incident.

7 Conclusion and Future Work

Our approach represents a first step towards representing security incidents automatically. A security engineer is required to simulate potential incidents by using the software system in a black-box fashion. We represent security incidents using UML Sequence Diagrams. We avoid representing all the methods that are invoked during the simulation of an incident (e.g., methods from classes that are part of the graphical user interface), which may not be relevant to security incidents. However, at the same time, we ensure the traceability of the security incident model with the software system components (e.g., methods and classes). This has the advantage to provide concrete guidance to the software engineers about where logging statements should be implemented and what information they should record. Our work is based on the assumption that security incidents are associated with access and/or modification of valuable information in the database. We also tailored the implementation of our approach to Java applications, using AspectJ. In

future work, we will apply our approach to other, more complex, software systems, considering other types of security incidents. We also aim to give a security engineer the possibility to edit the incident model including additional information relevant for logging purposes, such as more specific conditions that require to perform logging or additional information that should be recorded.

8 Acknowledgments

This work was supported by Science Foundation Ireland grants 15/SIRG/3501 and 13/RC/2094 and the EU H2020 CyberSec4Europe project grant 830929.

References

- [1] M. Andreessen, “Why Software is Eating the World,” p. 6, 2011. [Online]. Available: <https://a16z.com/2011/08/20/why-software-is-eating-the-world/>
- [2] D. Bayern, “Yesterday, Today, and Tomorrow 50 Years of Software Engineering,” *IEEE Softw.*, 2018.
- [3] C. Cimpanu, “FBI says cybercrime reports quadrupled during COVID-19 pandemic,” p. 3, 2020. [Online]. Available: <https://www.zdnet.com/article/fbi-says-cybercrime-reports-quadrupled-during-covid-19-pandemic/>
- [4] OWASP, “The Ten Most Critical Web Application Security Risks,” p. 25, 2017. [Online]. Available: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/
- [5] A. Taylor, “The 16 biggest data breaches of the 21st century,” p. 11, 2018. [Online]. Available: <https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html>
- [6] Marriott International, “Marriott Announces Starwood Guest Reservation Database Security Incident,” p. 2, 2018. [Online]. Available: <https://news.marriott.com/2018/11/marriott-announces-starwood-guest-reservation-database-security-incident/>
- [7] C. Cimpanu, “Marriott CEO shares post-mortem on last year’s hack,” p. 5, 2019. [Online]. Available: <https://www.zdnet.com/article/marriott-ceo-shares-post-mortem-on-last-years-hack/>
- [8] A. Chuvakin, K. Schmidt, and C. Phillips, *Logging and Log Management. The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*, 2013th ed. Syngress, 2013.
- [9] J. King and L. Williams, “Log Your CRUD: Design Principles for Software Logging Mechanisms,” in *Proc. 2014 Symp. Bootcamp Sci. Secur. (HotSoS’14)*. Raleigh, North Carolina, USA: ACM, 2014, p. 10.
- [10] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, “Learning to log: Helping developers make informed logging decisions,” in *Proc. - Int. Conf. Softw. Eng. (ICSE’15)*, vol. 1. Florence, Italy: IEEE/ACM, 2015, pp. 415–425.
- [11] R. Mckemmish, “When is digital evidence forensically sound?” *IFIP Int. Fed. Inf. Process.*, vol. 285, p. 13, 2008.
- [12] D. Alrajeh, L. Pasquale, and B. Nuseibeh, “On Evidence Preservation Requirements for Forensic-Ready Systems,” in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.* Paderborn, Germany: ACM, 2017, p. 10.
- [13] B. Endicott-Popovsky, N. Kuntze, and C. Rudolph, “Forensic readiness: Emerging discipline for creating reliable and secure digital evidence,” *J. Harbin Inst. Technol.*, vol. 22, no. 1, p. 9, 2015.
- [14] L. Pasquale, D. Alrajeh, C. Peersman, T. Tun, B. Nuseibeh, and A. Rashid, “Towards Forensic-Ready Software Systems,” in *Proc. 40th Int. Conf. Softw. Eng. New Ideas Emerg. Results (NIER’18)*. Gothenburg, Sweden: ACM, 2018, p. 4.
- [15] R. Rowlingson, “A Ten Step Process for Forensic Readiness,” *Int. J. Digit. Evid.*, vol. 2, no. 3, p. 28, 2004.
- [16] F. Rivera-Ortiz and L. Pasquale, “Towards Automated logging for forensic-ready software systems,” in *Proc. 6th Int. Work. Evol. Secur. Priv. Requir. Eng. (ESPRE’19)*. Jeju Island, South Korea: IEEE, 2019, p. 7.
- [17] B. Carrier and E. Spafford, “An Event-Based Digital Forensic Investigation Framework,” in *Proc. Fourth Digit. Forensic Res. Conf. (DFRWS’04)*, Baltimore, Maryland, USA, 2004, p. 13.
- [18] The Apache Software Foundation, “Log4j 2 API,” p. 4, 2020. [Online]. Available: <https://logging.apache.org/log4j/2.x/manual/api.html>
- [19] D. Oliveira, M. Rosenthal, N. Morin, K. C. Yeh, J. Cappos, and Y. Zhuang, “Its the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developers blind spots,” in *Proc. 30th Annu. Comput. Secur. Appl. Conf.* New Orleans, Louisiana, USA: ACM, 2014, p. 10.
- [20] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield, *Building Secure & Reliable Systems*, march 2020 ed. O’Reilly Media, Inc, 2020.
- [21] H. Assal and S. Chiasson, ““Think secure from the beginning”: A survey with Software Developers,” in *Proc. Conf. Hum. Factors Comput. Syst. (CHI 2019)*, ACM, Ed., Glasgow, Scotland, UK, 2019, p. 13.
- [22] G. Sindre and A. Opdahl, “Eliciting Security Requirements by Misuse Cases,” *Requir. Eng.*, vol. 10, no. 1, p. 12, 2005.
- [23] L.-C. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett, “Analysing Security Threats and Vulnerabilities Using Abuse Frames,” in *Proceedings Third Int. Work. Autom. Verif. Infin. Syst. (ETAPS’04)*, 2004, p. 18.
- [24] M. Almorsy and J. Grundy, “SecDSVL: A domain-specific visual language to support enterprise security modelling,” in *Proc. 23rd Aust. Softw. Eng. Conf. (ASWEC’14)*. Sydney, Australia: IEEE, 2014, p. 10.
- [25] B. J. Berger, K. Sohr, and R. Koschke, “Automatically Extracting Threats from Extended Data Flow Diagrams,” *Lect. Notes Comput. Sci.*, vol. 9639, p. 17, 2016.
- [26] P. Johnson, R. Lagerström, and M. Ekstedt, “A meta language for threat modeling and attack simulations,” in *Proc. 13th Int. Conf. Availability, Reliab. Secur. (ARES’18)*. Hamburg, Germany: ACM, 2018, p. 9.
- [27] W. Benghabrit, H. Grall, J. C. Royer, and M. Sellami, “Accountability for abstract component design,” in *Proc. to 40th Euromicro Conf. Ser. Softw. Eng. Adv. Appl. (SEAA’14)*. Verona, Italy: IEEE, 2014, p. 9.
- [28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” in *Proc. 11th Eur. Conf. Object-Oriented Program. (ECOOP’97)*. Jyväskylä, Finland: Springer, 1997, p. 11.
- [29] J. King, R. Pandita, and L. Williams, “Enabling Forensics by Proposing Heuristics to Identify Mandatory Log Events,” in *Proc. 2015 Symp. Bootcamp Sci. Secur. (HotSoS’15)*. Urbana, Illinois, USA: ACM, 2015, p. 10.
- [30] A. Chuvakin and G. Peterson, “How to do Application Logging Right,” *IEEE Secur. Priv.*, vol. 8, no. 4, p. 4, 2010. [Online]. Available: <https://www.computer.org/csdl/magazine/sp/2010/04/msp2010040082/13rUwIF6jg>
- [31] Oracle, “Processing SQL Statements with JDBC,” p. 5, 2019. [Online]. Available: <https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html>
- [32] Stack Overflow, “Java GUI frameworks. What to choose? Swing, SWT, AWT, SwingX, JGoodies, JavaFX, Apache Pivot?” 2013. [Online]. Available: <http://stackoverflow.com/questions/7358775/java-gui-frameworks-what-to-choose-swing-swt-awt-swingx-jgoodies-javafx>