Radboud University Nijmegen

Master Thesis Computer Science

A benchmark approach to analyse the security of web frameworks

Author:
K. Reintjes, BSc.
kreintjes@gmail.com

Supervisor:
Prof. dr. M.C.J.D. van Eekelen
marko@cs.ru.nl

Second reader:
Dr. ir. E. Poll
e.poll@cs.ru.nl

February 19, 2014
Abstract

Web frameworks often offer various security functionalities and protection mechanisms that developers can use to secure their applications. However, as it turns out, these frameworks themselves are not always that secure, which can have severe consequences. For example, one vulnerability in the Ruby on Rails framework was so severe that many web applications had to be taken off-line temporarily, among which the Dutch government’s authentication system DigiD. Also other popular frameworks have had several security problems in the past. Unfortunately, it appears there is no good way to analyse the security of these frameworks and detect vulnerabilities before they occur in released versions. We also note a significant lack of scientific research on the security of web frameworks. Therefore, the goal for this research was to find a general methodology to analyse the security of web frameworks. With this methodology it should be possible to detect vulnerabilities in any web framework, preferably in a (partially) automated way.

There are several challenges when trying to analyse the security of a web framework. These challenges make it hard to analyse the framework directly. Therefore we propose a benchmark approach. This approach uses a benchmark implemented in the target framework, which is analysed with well known dynamic web vulnerability scanners. The approach includes a general, framework-independent design for this benchmark. To use the approach, the benchmark needs to be implemented in the target framework. This can be seen as an instantiation of the benchmark for that framework. We then use dynamic web vulnerability scanners on this implementation to analyse the security of the framework. A vulnerability discovered in the benchmark implementation could indicate a vulnerability in the framework.

During this research we developed our approach and designed the required benchmark for SQL injections and XSS vulnerabilities. We also tried the approach in practice, by applying it on the Ruby on Rails web framework. We implemented the benchmark in this framework and analysed it using two dynamic scanners, Arachni and W3af. Using our approach we discovered five vulnerabilities, of which three were completely new. In this thesis we present our approach and the design of the benchmark. We discuss the benchmark implementation in Rails and the results of the analysis. Finally, we evaluate the approach and results, and present potential improvements and other ways to analyse the security of web frameworks. We conclude that our approach is indeed capable of analysing web frameworks for security vulnerabilities, but is not perfect either, since it has several weaknesses.
Acknowledgements

I would like to thank several people for their help with my research and/or my thesis. First of all, my supervisor, Marko van Eekelen, for suggesting to widen the topic from “Ruby on Rails” to “web frameworks in general”, as well as for his help during the research and his constructive feedback on my thesis. Secondly, Erik Poll, for acting as second reader for this thesis.

Furthermore, I thank Christiaan Thijssen for his useful advice and helpful discussions, as well as reviewing the complete final draft version of this thesis. I also would like thank Arjan Diepenbroek for reviewing several chapters of the thesis, and of course for answering all my questions about English grammar and spelling. For the latter, I also thank Floris de Lange.

Finally, I wish to thank my parents for their general support and advice throughout the complete project.
Chapter 1

Introduction

1.1 Background and motivation

Web applications are becoming more and more popular and many software developers are rewriting their software as a web application and offering it to their clients as SAAS (Software as a Service). This has as main benefit that the application is easy accessible. Clients do not have to install and configure the software any more, which sometimes involves the purchase of expensive special servers. Web applications are written in a programming language, which often is a scripting language such as PHP\footnote{http://www.php.net/} or Ruby\footnote{http://www.ruby-lang.org/}.

However, since many web applications possess similar functionalities and developers do not want to reinvent the wheel every time they write a new application, a framework is often used.

Web application frameworks, or simply web frameworks, support developers in creating powerful web applications in a quick, efficient and consistent manner. A web framework can be seen as an extension on top of the programming/scripting language, that offers many common functionalities that developers can directly put to use. For example, they often provide libraries for database access, session management, authentication and authorisation, and more. All of which are very common and useful in web development. All this ready to use functionality can save developers a lot of time. Furthermore, web frameworks often use and promote certain design patterns and principles, which leads to nicer, better structured and more robust code. For example, many frameworks use the model-view-controller (MVC) pattern, that nicely separates the data model, business rules and user interface.

However, not only speed and beauty of the code are important factors to consider when building web applications, security is also very important. There are many common vulnerabilities \cite{Pro13}, such as Cross-Site Scripting (XSS), SQL injections and Cross-Site Request Forgery (CSRF). Those vulnerabilities threaten web applications on a large scale. There also have been successful attacks on large web applications in the past, leading to theft of personal data, disablement of the application (Denial of Service), or worse. In fact, security is believed to be too critical to leave up to individual programmers \cite{SS02}. Fortunately, web frameworks can help with this as well. They usually provide several built-in functions and mechanisms to provide protection against common vulnerabilities. These mechanisms make it easier for developers to build secure applications. Developers can save time because they do not have to implement the security functionalities themselves. They can rely on the mechanisms provided by the web frameworks, which are often a lot more secure than what they can build themselves. Furthermore, sometimes the security functionality is even applied “automatically” so the developers do not continuously have to worry about it themselves\footnote{An example of this is the automatic XSS sanitization as discussed in \cite{WSA+11}.}. This is particularly useful in very complex web applications that consist of ten thousands lines of code or more. In these applications, a developer could easily forget to implement the security measures for a new feature, leading to a hard to spot security vulnerability.

Unfortunately, as it turns out, these frameworks themselves are not always so secure. For example, in one particular framework, Ruby on Rail\footnote{http://rubyonrails.org/} there recently were several critical security bugs discovered. As a concrete example, the CVE-2013-0156 vulnerability\footnote{https://groups.google.com/forum/#!topic/rubyonrails-security/61bgv38jwTQ/discussion} discovered in January 2013, posed a major risk for all Ruby on Rails web applications. This vulnerability allowed attackers to bypass authentication
systems, inject arbitrary SQL, inject and execute arbitrary code or perform a Denial of Service attack. So basically, every of the five hundred thousand web applications running Ruby on Rails was completely at risk. However, not only the Ruby on Rails framework had security problems in the past. Other frameworks, such as the Zend framework, the Django framework and the CakePHP framework have (had) significant security vulnerabilities as well.

Vulnerabilities in web frameworks are extremely dangerous, since they can instantly threaten thousands of independent web applications. Because the vulnerability targets so many applications at once, it is likely that easy to use exploits will be developed soon. Even if these vulnerabilities are quickly fixed, it could take some time before all the applications are patched. Furthermore, it costs many companies and individual people time and/or money to urgently patch their applications. The CVE-2013-0156 vulnerability is a clear example of the trouble a single vulnerability in a web framework can cause for many individuals. For these reasons, it is preferable to prevent these vulnerabilities from occurring at all (ending up in a production release of the framework), rather than discovering and fixing them afterwards. In this thesis we will propose and evaluate a general methodology to analyse the security of web frameworks, which can be used to discover vulnerabilities.

1.2 Analysing the security of web frameworks

Finding such a general methodology to analyse the security of arbitrary web frameworks is not trivial. There is a lack of scientific research into this topic, which might be explained by the difficulty of this problem. There are of course many publications about web application security in general, for example discussing common vulnerabilities of web applications or how to detect/prevent these. There are even many automated tools to analyse web applications for common security vulnerabilities. Examples of these techniques include dynamic web vulnerability scanners, symbolic security analysis and taint analysis. With regard to web frameworks however, there is much less available and it appears there is no general method on how the security of a web framework can be analysed (yet). We believe this is mainly because of three challenges to be considered when analysing the security of web frameworks:

1. First of all, web applications and frameworks are often not written in a formal language. Therefore we cannot simply use formal proofs to prove certain properties of the framework, such as that it is secure and does not contain vulnerabilities.

2. Analysing the security of a framework is hard, because it is not yet clear how the parts of it are going to be combined. Web frameworks are partially a collection of useful tools and mechanisms, which developers can use at their will. It is up to the developer to decide what he wants to use and what parts he combines. In a finished web application it is clear how the several parts are put together. Therefore it is somewhat easier to check if this can lead to security vulnerabilities, and there are even automatic tools to do so. With a framework however, it has yet to be determined which parts are going to be used together and if this can lead to security issues or not. This challenge could be one of the reasons for the lack of related work in this topic.

3. There can be huge differences between separate web frameworks. They can have a different overall architecture, follow different design principles and/or could differ in the programming language they run on. Because of this, it will likely be hard to create a general method which can be used to analyse the security of all web frameworks. This is especially the case when the analysis is automated.

Considering these challenges, we came to the idea of using a benchmark consisting of test applications, that can be used to analyse the security of the framework. The rationale is that it is hard to verify the

---

[4] html
[6] html

As was the case for the CVE-2013-0156 vulnerability in Ruby on Rails. Only a day after disclosure, the first exploits were already released. [http://ronin-ruby.github.com/blog/2013/01/09/rails-pocs.html](http://ronin-ruby.github.com/blog/2013/01/09/rails-pocs.html)
security of the framework directly, so we analyse it through an extensive benchmark, which consists of one or more test applications written in the framework. This way we address the second challenge. We use the already available tools to analyse the security of web applications on a very elaborate benchmark to analyse the security of the framework. To address the third challenge, the benchmark will consist of a high level, abstract design of the applications, rather than the actual applications (implementations in the frameworks) themselves. This way the benchmark can be used for any framework, it just has to be implemented for it. This can be seen as an instantiation of the benchmark for a particular framework. We will discuss and motivate the idea of using a benchmark further in chapter 4.

1.3 Research goal

The main goal of this research is to propose a general method to analyse the security of web frameworks. As discussed above, we will use a benchmark consisting of test applications for this. Therefore, we need to determine the effectiveness of analysing the security of web frameworks using such a benchmark. This leads to the following research question:

Can we analyse the security of web frameworks in a general and uniform way by using a benchmark of test applications?

However, in order to determine this, we should determine what such a benchmark should look like, i.e. establish the design of the benchmark. Furthermore, we should establish a way to test the benchmark, in order to detect vulnerabilities in the framework. These two parts together form the proposed methodology for analysing the security of web frameworks and shall be elaborated in chapter 3. Besides from only proposing an approach, we will also directly put it to use and evaluate it. To do this, we have chosen one web framework, namely Ruby on Rails. We do this by implementing the benchmark in Ruby on Rails and then use it to analyse the security of Ruby on Rails.

1.4 Research contributions

Following the research goal stated above, this research has four main contributions:

1. A general methodology using a benchmark to analyse the security of web frameworks:
The most important product of this thesis is a general method to analyse the security of arbitrary frameworks. This method uses a benchmark consisting of test applications in order to overcome the two important challenges when analysing the security of frameworks. Therefore, this product consists of two parts: the design of such a benchmark and the approach of how the benchmark can be used to analyse the security of web frameworks.

2. The implementation of the benchmark in the Ruby on Rails framework:
The second product of this thesis is an implementation of the benchmark in the Ruby on Rails framework. We used the benchmark design to actually built the benchmark applications in Rails. These benchmark applications can be used to analyse the security of the current version and possibly future versions of the Rails framework.

3. The security analysis of the Ruby on Rails framework:
After we have developed the Ruby on Rails implementation of the benchmark, we have used it together with our approach to analyse the security of the Ruby on Rails framework. We discuss the security of the Rails framework and indicate possible new vulnerabilities.

4. The evaluation of the approach and benchmark:
The final product of this thesis is the evaluation of the proposed method and the benchmark itself. During the development of the benchmark, its implementation in Rails and the security analysis that followed, we gained knowledge about the practicality of our approach. We use this knowledge to evaluate our approach, the results and the benchmark and discuss strengths and weaknesses.

11 We will elaborate our choice for Ruby on Rails later in chapter 3.
1.5 Organisation of this thesis

This thesis is outlined as follows: in the next chapter we give a general introduction into web frameworks and discuss the common security functionalities they provide. In chapter 3 we give a detailed introduction into the Ruby on Rails framework. We discuss Rails’ security functionalities, as well as recent vulnerabilities discovered in them. Then, in chapter 4 we present our approach to analyse the security of web frameworks. Furthermore, we present the design of the benchmark and its modules. After that, we discuss the implementation of the benchmark in Rails (chapter 5). In chapter 6 we present the options (tools) available to us for the analysis. We use these tools to analyse the Rails benchmark implementation and with that the framework itself. The results of this analysis are presented in chapter 7. We conclude this thesis by evaluating our approach, the benchmark and its results (chapter 8). In this evaluation we discuss the strengths and weaknesses of our approach, present potential improvements and give directions for future research.
Chapter 2
Web frameworks and security

This chapter provides some background information on web frameworks and related security issues. We discuss what web frameworks there are, what similarities they have and in what they differ. After that, we discuss common vulnerabilities for web applications and how frameworks help protect against these. This will help to determine what kind of vulnerabilities our analysis method should detect.

2.1 Web frameworks

A web application is an application that is accessed by users over a network, such as the Internet. Web applications usually interact with a database, allowing a user of the application to (indirectly) act with that database as well. They are written in a programming language chosen by the developer. Often web applications are written in dynamic scripting languages. Examples of programming languages often used for web applications are PHP, ASP.NET, Java, Ruby, Python and Perl. However, web applications are usually not written directly in such a programming language, but rather in a web framework. This is because, from a developer’s point of view, many web applications perform similar activities and it does not make sense to implement the same functionalities every time. Web frameworks provide libraries that implement many of such tasks, thus easing the development of web applications.

Originally, a web framework could be seen as a collection of libraries aimed at aiding web developers. These libraries are written in the same programming language as the application. The developer can include whatever functionalities from the framework he needs. However, nowadays so-called “full stack” frameworks are more common. These frameworks are not simply a collection of libraries, but can be seen as an extension of the programming language. Of course the frameworks still provide many common ready-to-use functionalities, but they also do more. They form a cohesive software stack in which developers build their applications. The developers are somewhat limited in their freedom. They have to follow the rules, i.e. the design patterns and principles, of the framework. They should not simply write the web application in the programming language they chose and only include some needed functionalities, but are more or less forced to work completely in the framework. This restriction in freedom comes with the following benefits; web frameworks can offer even more ready to use functionalities and web applications can more easily be written in a nice and consistent manner. If really desired it is often still possible to break the frameworks way of doing things, although this requires extra work. Fortunately, this is often not necessary.

In this thesis we focus on these full stack frameworks, since they often also provide the most functionalities for securing web applications. Furthermore, in many of these frameworks, the security functionalities are even applied automatically. Full stack frameworks can differ on many things, but usually they also have similarities. We will now give an overview of the most important properties of web frameworks. We also explain on what kinds of web frameworks this research focuses and what not. After that we present a list with examples of widely used web frameworks.
2.1.1 Architecture

The first important property of a web framework is the kind of architecture it uses. This is split up in two parts, the first being the tier(s) the framework operates on, the second being the architectural pattern it follows. We explain the differences and on what kind of frameworks this research focuses.

Three tier organisation

Web applications are usually structured in a so called three-tier organisation, which consists of three physical layers: client, application and database. The application runs on a (web) server and contains the business logic. It interacts with a database, usually an RDBMS (Relational Database Management System), which is used to store persistent data. In web applications the client is a web browser, that runs the HTML generated by the application. The client communicates and acts with the application through the HTTP interface. The client can perform actions on the application, which in turn change the state of the database. So the client interacts with the database through the web application.

Web frameworks are used to build the applications, so they are mainly used for the second tier (application), but sometimes they can have some small overlap with the first (client) or third (database) tier. There are even frameworks/programming languages, operating on all three tiers. An example of such a language is Links [CLWY07]. However, most commonly the web framework only operates on the second tier. For this reason, our research focuses on web frameworks operating (mostly) on the second tier.

Relatively new are so-called client-side (web) frameworks, usually written in JavaScript and operating on the first tier (client) [OPKJ10]. These frameworks allow the creation of “rich” internet applications, having an extensive (graphical) user interface. Most of the work and processing is performed on the client-side, sometimes without having a server-side back-end at all, except for the web server serving the application’s code as static content. Sometimes these frameworks are also called web frameworks. However, we do not consider them as such and do not include them in this research. The reason for this is that they do not contain security functionalities. If applications written in these frameworks need to share data between users, then they would need a server-side database (third tier) to do so. Access to this database should be secure and vulnerabilities such as SQL injections should be prevented. Therefore, the client-side frameworks should implement security protection mechanisms. However, since these frameworks are operating on the client-side (as JavaScript code), the client can manipulate them. This means that a potential attacker (client) can modify or remove the protection mechanisms at will, leaving the database open for attack. For this reason, client-side web applications can never communicate with the database directly. Instead, they do this through an interface (API) residing on the server. This means these applications always need a “normal” web application on the second tier implementing this API. This normal application is also responsible for implementing the security functionalities and protection mechanisms. Because these client-side web frameworks always need a “normal” server-side web application to deal with security issues, we further ignore them and focus solely on server-side (or tier two) web frameworks.

Architectural pattern

Web frameworks usually follow some kind of architectural pattern (or at least, they should) and they promote or even enforce the use of this pattern in the web applications as well. The pattern determines for example how the code is structured, what is put where and what components may call/use each other (directly). The most used architectural pattern for web frameworks is the Model-View-Controller (MVC) pattern. It is for example also used by Ruby on Rails. Other architectural patterns used by web frameworks, are for example the Page Oriented architecture, Event Driven architecture or Model Driven architecture. It is also possible that a framework does not follow a clear architectural pattern at all (or only uses it for specific parts), although these frameworks usually are not nicely structured and lead to bad readable and maintainable code. We will only consider the MVC pattern further, since it is by far the most common for web frameworks.

MVC pattern

The MVC pattern separates complex programs, such as web applications, into three units with different responsibilities: the data model (Model), the data representation (View) and the application logic (Controller). The model contains the business logic and could interact with some storage device, such as
The view presents the data to the user. The model and the view are glued together by the controller. The controller determines what actions can be performed by the user, then performs these actions using the data model and presents the result to the user using the view. Separating the program this way has several benefits, such as that it modularizes the code and promotes code reuse. Another benefit is that it makes sure that changes in the representation of the data do not necessarily influence the data model or vice-versa. This also allows the developer to use multiple different views for the same model. This is particularly useful for web applications, since it allows to create multiple interfaces for the same model. For example you could create a normal HTML web page for human users and a JSON web service for remote applications.

Web frameworks that use the MVC pattern usually force the developer to separate their applications this way as well. Often they also specify a certain format for each unit and location to store the files. They could also supply helper functions and even enable some forms of interaction between the different units, or external entities, such as the database, automatically.

MVC in web frameworks can be implemented in two ways: following a push-based approach or a pull-based approach \[LR01\]. In the push-based architecture (also called action-based), the controller has several actions which perform the requested processing (requested by the user) and then push the data into the views to display the result. In the pull-based architecture (also called component-based), the framework starts with the view and then pulls the needed data from the controllers. In this architecture multiple controllers can be used in a single view. The most common however, is the push-based architecture, which is for example also used by Ruby on Rails.

### 2.1.2 Features

The second property in which web frameworks could differ is the features they offer to the web developer. Not all web frameworks offer the same features. For example, while most frameworks offer some kind of database interface, caching functionalities are much less common. We will now present the most common features web frameworks could have and explain them shortly.

- **Database interface**: many frameworks offer an interface to perform database operations. However, what they offer exactly and how this interface is implemented can differ significantly. Since the database interface is of particular interest for our research (because of possible SQL injections), we discuss this feature in more detail later in section 2.1.3.

- **Template engine**: a template engine combines web templates and data from various sources to form a finished web page. Benefits of using a template engine are that it separates view code (often HTML) from the data processing and application logic code. For this reason a template engine is a must for MVC frameworks. Furthermore, it allows to nicely structure the views in coherent subparts and reuse these parts in other places to prevent code duplications. Many frameworks offer one or more template engines. The template engine can be of particular interest for our research, since they can provide mechanisms to (automatically) prevent output injection attacks, such as Cross-Site Scripting \[6\].

- **Form validation**: frameworks often supply form validation tools, to validate form input values and possibly also automatically collect and display validation errors. Some frameworks even come with a complete form builder, which can be used to build a form, specify the validations and provide nice error feedback to the user. For example, Ruby on Rails includes a form builder that can create and fill drop-down fields, automatically highlight fields with errors and preset default values.

- **Caching**: caching can help to reduce the bandwidth usage, server load and response times. Since web applications should be as fast as possible, and may not use too much server resources, many web frameworks include caching mechanisms. Caching can be implemented on various levels of the execution process. What mechanisms are provided and on what levels these are implemented depends on the framework.

---

\[5\] For the latter the framework usually should also follow another design pattern, namely Convention over Configuration (CoC) \[BK07\], which for example is also used in Rails.

\[6\] More on this and other security threats in section 2.2.
Chapter 2. Web frameworks and security

- **Routing**: as discussed earlier, MVC frameworks provide a nice separation of the data model, the data representation and the application logic. However, web applications are basically web pages responding to HTTP requests. So we need some way to let a particular HTTP request, for a particular URL, trigger a specific piece of the application logic. In other words, to trigger a specific piece of code in some controller. This is were routing (also called URL mapping) comes into effect. The routing mechanism maps URLs/HTTP request to particular controller parts and possibly also vice-versa.

- **Internationalisation**: many frameworks provide mechanisms and tools to simplify internationalisation (I18n) and localisation (L10n). Often the internals of the framework are already completely internationalised and many language files (locales) are provided.

- **AJAX and JavaScript**: AJAX is becoming more and more important in modern web applications. Many frameworks offer functionalities to ease the usage of JavaScript, often in combination with a JavaScript library such as jQuery. For example, Ruby on Rails integrates jQuery by default, and as of version 4.0 also includes a new feature called Turbolinks, which will basically translate every request in an AJAX request to improve performance and responsiveness.

- **Web services**: originally web frameworks where intended to create web applications serving web pages to human users. However, as the web becomes more integrated, web applications often have to communicate with other, external systems as well. This is what web services are meant for. A web service is a piece of the application which can be used by other systems to perform actions or read data. Many web frameworks provide tools for creating web services. For example, Ruby on Rails, provides a mechanism to make any web page also available as a web service, using JSON or XML.

- **Testing**: many frameworks provide several tools to ease with (automatic) testing. Frameworks can include tools to specify and (automatically) check unit tests, functional tests and integration tests. Some frameworks, including Ruby on Rails, even promote the use of these tools explicitly and encourage Test Driven Development (TDD) \[Bec02\].

- **Security**: a feature particularly interesting for our research is of course security. Frameworks often provide several tools to help prevent common attacks, such as SQL injections, Cross-Site Scripting and Cross-Site Request Forgery. They could also offer implementations for some security functionalities, such as authentication and authorization solutions. Since the security feature is so interesting for us, we discuss it in more detail later in section \[2.1.4\].

2.1.3 Database interface

Almost all web frameworks offer some kind of database interface to perform database operations. However, what they offer exactly can differ significantly. Some frameworks only provide the communication interface, where you still have to write the queries yourself. Other frameworks also provide query builders that allow you to easily build queries for common operations. These query builders could also apply sanitization on the input data automatically. Some frameworks try to reduce the work for the developers as much as possible and provide many helper functions and even perform common database tasks automatically. For example, in Ruby on Rails, to save an object to the database, you simply call the save method. Rails then automatically generates the INSERT or UPDATE query, applies sanitization on the input data, and executes the query for you \[BK07\].

Also the way the database interface is implemented can differ significantly. The object-oriented/MVC frameworks that provide a more extensive database interface use an Object-Relation Mapping (ORM), that maps objects (models) to database rows and vice-versa. However, there are many different types of ORMs and many different ways to implement them. Some examples of common ORMs are: Table Data Gateway, Row Data Gateway, Active Record and Data Mapper \[Fow02\]. Ruby on Rails uses the Active Record pattern, which means that an object (model) represents a single row in a database table, encapsulates the database access, and adds domain logic on that data. The Active Record pattern is the main cause for the simplicity of saving objects to the database.
2.1. Web frameworks

Most interesting for this research however, are the security functionalities (e.g. sanitization) it supplies. These functionalities should be included in the benchmark, such that they can be analysed. It should be verified that these functionalities prevent SQL injections. Also the way the functionality is implemented, including the type of ORM the framework uses, is important to us. This determines how the tests should be implemented in the benchmark. For example, in a framework that does not automatically apply the sanitization of user input data, we should make sure that the sanitization helpers are indeed being included in the benchmark.

2.1.4 Security

Many web frameworks offer various security features. These features can be split in two categories:

1. The implementation of security functionalities common in web applications used to achieve certain security goals. Two examples of such functionalities are authentication and authorisation. Many web applications offer some private section, for which users need to register and login before they can access it. This requires the implementation of some authentication mechanism, and probably also an authorization framework to determine which users may perform certain actions. Since some of these security functionalities are so common in web applications, some frameworks offer directly available implementations that developers could use. Ruby on Rails does not offer many of these implementations\(^8\) but other frameworks do. For example, CakePHP\(^9\) includes a complete, ready-to-use, authentication and authorization solution.

2. Protection mechanisms (tools or methods) to protect against common security vulnerabilities in web applications. Examples of these common vulnerabilities are SQL injections and Cross Site Scripting. Developers can use these protection mechanisms manually to secure their applications. Often, the protection is applied automatically by the framework, leading to *security by default*. This makes the existence of isolated security vulnerabilities, caused by a developer forgetting to apply the protection, much less likely.

The first category is not of great importance for this research, and we have chosen to further ignore it. The reason for this is the difficulty of testing the correctness of implementations of security functionalities. The desired behaviour and the requirements are not always clear and could differ between frameworks. Furthermore it is hard to write a formal model for it and (automatically) check the correctness of the implementation against this model. At last, the set of functionalities included by the various frameworks could differ significantly and some frameworks do not even include any of these functionalities at all.

The second category, is of great importance. Web developers can use the protection mechanisms supplied by the framework to secure their applications. However, for this to work, it is important that the tools provided by the frameworks are secure themselves. This is exactly the point this research focuses on. The benchmark should analyse the security of these protection mechanisms, i.e. verify that they do not contain vulnerabilities. Vulnerabilities in these protection mechanisms mainly happen for two reasons:

1. The base protection mechanism itself is not secure, e.g. it does not cover all the subtleties of the vulnerability, allowing more sophisticated attacks. The benchmark should therefore verify that the protection mechanism work correctly and achieve their goals.

2. The protection mechanism is not always applied, e.g. it is simply forgotten for a certain attack point. This creates a false sense of security, i.e. it lets the developer believe he does not need to perform any protection himself, while this is not the case. The benchmark should therefore also make sure that protection is applied everywhere it is needed.

Since this second category is of such importance to our research, we will discuss it in more detail in section 2.2. There, we list and explain all security mechanisms usually provided by web frameworks.

---

\(^8\)Although Ruby on Rails can be extended with plug-ins, so-called gems, that do offer these implementations. One example of this is Devise, a widely used authentication solution (https://github.com/plataformatec/devise).

\(^9\)http://www.cakephp.org
2.1.5 Examples of web frameworks

We will now give some examples of commonly used web frameworks for a few programming languages.

PHP

PHP by itself already resembles much of a web framework. In the first place, PHP is a programming language specifically designed for the web. It is easy to setup PHP with a web server, and build dynamic web pages with it. For many other languages, more effort is required. Furthermore, PHP handles some common tasks in web development, such as output handling and parsing of request data and making it easy accessible. It also comes bundled with an extensive set of modules and ready-to-use functions, for many tasks, varying from database communication to image processing.

PHP works well for building small websites with some simple dynamic features. However, it is cumbersome to build large applications in it. Therefore, despite the extensiveness of PHP itself, there are many frameworks developed on top of it, that even further extend its functionality. Some examples of widely used frameworks are: CakePHP, Prado, Yi framework, and Zend Framework. All of them are MVC frameworks, of which CakePHP is push-based, while the others are both push and pull-based. They supply a wide-range of features, including a template engine and complete database interface, that makes it easier to perform or automates many common tasks.

ASP.NET

Just like PHP, ASP.NET also resembles much of a web framework by itself. However, again it is not recommended to develop large application in it directly. Therefore, there are some frameworks built on top of it. Examples are OpenRasta, MonoRail and ASP.NET MVC. The MonoRail framework is particularly interesting to us, since it is inspired by Ruby on Rails. Just as Rails, it comes with a database interface following the Active Record pattern. It includes also many other features similar to those of Rails.

However, the most popular ASP.NET framework is not MonoRail, but ASP.NET MVC. Clearly it is an MVC framework, implementing the push-based architecture. It comes bundled with a complete database interface, along with several other features such as a template engine, form validation framework, testing framework and internationalization support. It also supplies several security features, such as an authentication and authorization solution, but for example also offers protection against CSRF attacks, about which we will tell more later.

Ruby

There are not that many web frameworks for Ruby. Ruby on Rails is by far the most popular one. Since this thesis explicitly focuses on it, we will discuss it in detail in chapter 3.

Three other Ruby web frameworks are Merb, Camping and Sinatra. Merb was a web framework very similar to Rails, but only included the most essential core functionalities, leaving the rest to plug-ins. It existed for quite some time alongside Rails, but as of August 2010 its best ideas were merged into Rails and Merb is no longer under active development.

Camping is a so-called micro framework. It is less than four kilobytes large and consequently not very extensive. It is based on the MVC push architecture and it comes bundled with a template engine and (Active Record) database interface. It does however not come with many security features.

---

10 This list is by far not exhaustive. For a more complete list of web frameworks and their properties, see [http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks](http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks). This will also give an impression of the large variety of frameworks there is.

16 [http://www.asp.net/mvc](http://www.asp.net/mvc)
17 [http://www.merbivore.com](http://www.merbivore.com)
18 [http://camping.io/](http://camping.io/)
Sinatra\(^{19}\) is also a MVC push-based framework, although it follows this architecture less strict. It does not bundle a database interface, but allows developer to choose their own ORM through plug-ins. It does however bundle a template engine and some security features, although less extensive than Rails’.  

### Other languages

Besides the programming languages mentioned above, there are of course also web frameworks for other, yet unmentioned languages. For example, there are many web frameworks for Java (e.g. Google Web Toolkit\(^{20}\)), Perl (e.g. Catalyst\(^{21}\)) and Python (e.g. Django\(^{22}\)). There are too many frameworks to discuss in detail, so we will leave it at this.

#### 2.2 Common security functionalities in web frameworks

We now list and explain the most common security functionalities and protection mechanisms offered by web frameworks. We first explain the vulnerability it should protect against and then explain how the protection mechanism usually works. Some functionalities do not protect against a clear single vulnerability. In that case, we will simply explain the purpose of the mechanism. Some protection mechanisms could be implemented in multiple ways, different for each framework. Then we will discuss the most common ways.

##### 2.2.1 Protection against SQL injections

**Vulnerability**

One of the most harmful vulnerabilities for web applications is the SQL injection (SQLi) attack. A web application is vulnerable to SQL injection attacks, if it uses unvalidated user input in the construction of an SQL query \(^{ST13}\). Consider for example an application with a login form. The web application usually verifies the entered credentials by constructing and executing a query like this:

\[
\text{SELECT } * \text{ FROM users WHERE username = '}<\text{username}>' \text{ AND password = '}<\text{password}>' \text{'}
\]

If the input of the password field is used directly in the query without any validation or sanitization, then an attacker could abuse this. If for example he enters the string ‘OR 1=1 LIMIT 1-- as username and leaves the password empty, then the resulting query becomes:

\[
\text{SELECT } * \text{ FROM users WHERE username = '' OR 1=1 LIMIT 1-- AND password = ''}
\]

The \textit{AND password = ''} part of the query is completely ignored because of the comment characters --. Since 1 always equals 1, the query will return the first user in the database, which is likely the administrator.

SQL injections are considered very harmful. They are the number one vulnerability in the OWASP Top 10 in both the 2010 \(^{Pro10}\) and 2013 \(^{Pro13}\) edition. SQL injections can have disastrous consequences. It could lead to exposure of confidential data, but also to the alteration or destruction of data. It is even possible to destroy complete databases. Fortunately, this type of vulnerability is nowadays well known, and SQL injection vulnerabilities are becoming less likely. However, a web developer should still be aware of SQL injections and mistakes are easily made, certainly when not using a centralized or automatic protection solution.

**Detection**

SQL injections are usually found by sending special SQL characters as input to the application and look for error responses indicating an SQL syntax error. Sending input to the application can be done in various ways: through HTML form fields, as part of the GET query string, through cookie values, as part

---

\(^{19}\)http://www.sinatrarb.com/
\(^{20}\)http://www.gwtproject.org/
\(^{21}\)http://www.catalystframework.org/
\(^{22}\)https://www.djangoproject.com/
of the URL, et cetera. All these ways should be tried. Usually simply sending a single quote character is enough. If the application is vulnerable, then this leads to an extra quote in the query, that is never terminated, thus resulting in a parse error. Of course, this only works if the application indeed reports SQL errors.

If the application does not report error messages, then other techniques should be employed. These are the so-called blind detection techniques. One method is to try timing attacks, by sending sleep statements to the SQL server. Another one is to construct two queries that should lead to different responses if the application is vulnerable. This is called differential analysis. An attacker could for example try sending both strings ‘AND 1 = 1’ and ‘AND 1 = 2’. If the application is indeed vulnerable, then the first attempt will display one or more results, while the second will display none. All mentioned detection techniques can also be performed automatically.

Protection

As said before, SQL injections occur because unvalidated user input is used to construct queries. Therefore, to prevent SQL injections, user input should be validated or not be used directly in SQL queries. Three common ways to prevent SQL injections are:

1. Verify that the input is as expected. The developer determines a list of allowed values and matches the input against these values. This is a whitelist approach. In case the input is more dynamic, then it can be matched against a pattern or regular expression, instead of fixed values.

2. Sanitize the input. Sanitizing the input is the process of escaping or removing all special SQL characters, such as the single quote (‘) character. After proper sanitization, the input should be safe for use in the SQL query. This is a blacklist approach.

3. Use prepared (or parameterized) statements. With prepared statements, the user input is not used in the SQL query directly, but is passed to the database management system separately from the query statement. This way the database knows these values are parameters, and cannot change the query statement, effectively preventing SQL injections. As a bonus, prepared statements can also have a performance increase, due to possible query optimization.

Web frameworks usually apply method two or three. The reason for this, is that they do not know what the expected input is, and thus cannot create a whitelist of allowed inputs. The frameworks can however supply sanitization helpers or a database interface that allows the use of prepared statements. In this case, the developer should use these protection mechanisms manually, e.g. he should apply the sanitization helper on the user input before using it in the query. However, there are also frameworks that apply the SQLi protection mechanisms automatically for you. Usually these are the frameworks with the more extensive database interface, i.e. the frameworks with a complete ORM using query builders, so developers do not have to write their own SQL. These query builders will then automatically apply sanitization on the input values or use prepared statements.

2.2.2 Protection against other injection attacks

Besides SQL injections, there are also other injection attacks. Four examples are Command Line injection, Code injection, XPath injection and LDAP injection [SW06 SP07]. They work basically the same as SQL injections: they occur because unvalidated user input ends up in some interpreter. They also have similar protection techniques, using one or more of the three techniques described for SQL injections, but then with some modifications.

Web frameworks could also offer these protection techniques as a feature. Almost all web frameworks offer some sort of protection against SQL injections. However, protection against other injection attacks is much less common. The reason for this, is that these vulnerabilities are also much less common than SQL injections. Almost all web applications use a database system to store data, but there are much less web applications that use user input in a system call (which could cause a command line injection) or use XPath or LDAP.

We will now shortly describe some other common forms of injection attacks and the protection mechanisms web frameworks could offer against these attacks.
2.2. Common security functionalities in web frameworks

Command Line injection

The web application uses user input to execute system commands, usually called with functions as `system` or `exec`. An example could be a web application that tries to ping a host, where the host is entered by the user: `ping <host>`. The attacker could enter `google.nl; rm -rf /` which will cause the server to delete all files.

Web frameworks achieve protection against command line injection usually by using a technique similar to prepared statements. The command and the arguments are separated from each other and the framework will encapsulate the arguments and make sure that the content of those arguments cannot break the encapsulation. Continuing the example above, the call would be changed from `system("ping <host>")` to `system("ping", <host>)`. The framework translates this to `ping "<host>"` and makes sure the host argument cannot break the encapsulation by adding extra quotes.

Code injection

Dynamic interpreted programming languages, such as Ruby and PHP, often have the possibility to perform dynamic code evaluation/execution, usually with a function called `eval`. Web applications written in such language could contain code injection vulnerabilities. A code injection vulnerability occurs when the web applications uses user input in the code evaluation function. An attacker could use this to insert arbitrary code in the web application, for example to bypass authentication systems or execute SQL queries.

Protecting against code injection is hard. All input is seen as code, and there is no way to filter all constructs that could possibly do harm. Therefore, using user input in dynamic code evaluation should be prevented as much as possible. It is considered a bad coding practice, since it creates the possibility for code injection attacks, while it is often not necessary to use the input in the evaluation or even use code evaluation at all. If despite this user input is still used in code evaluation, then the only way to make it secure, is to filter the input using a whitelist. Because of this, (automatic) code injection protection mechanisms in web frameworks are very rare. They cannot determine in advance what constructs to allow and what not, since they do not know the intention of the evaluation. Furthermore, what construct are safe to use, could be context-specific. Web frameworks will likely discourage the use of (user input in) code evaluation at all, rather than supplying protection techniques.

XPath injection

Same as SQL injection, only this time the target is the XPath interpreter, used to query XML documents. The protection techniques are also the same. For example, a web framework could offer sanitization helper functions or use precompiled XPath (a form of prepared statements for XPath). The protection could be applied automatically, if the framework offers a complete interface with expression builders, just as for SQL. However, this is much less common than is the case for SQL.

LDAP injection

Also very similar to SQL injection, but this time the target is an LDAP interpreter, used to query an LDAP server. LDAP injections are usually prevented using a whitelist or blacklist approach. The use of prepared statements appears to be less common. Web frameworks could offer protection mechanisms to prevent LDAP injections, although this is also much less common than for SQL injections. If they offer a protection mechanism, then it is usually in the form of sanitize helper functions that remove or escape special characters.

2.2.3 Cross-Site Scripting protection

Vulnerability

Cross-Site Scripting (XSS) is a vulnerability that allows an attacker to inject client-side script code, such as HTML or JavaScript, into the web pages of the application. This client-side code is then executed for other visitors of the website, leading to various security threats. An attacker

\footnote{Besides code injection attacks due to evaluation of code, there is also another form of code injection attacks, namely file inclusion. We will treat this form separately later in section.}
could for example show false content or steal cookie data, among which the session data, which can be used to impersonate other users. XSS is also considered very harmful, taking the third position in the OWASP Top 10 of 2013 [Pro13].

XSS vulnerabilities occur when user input is displayed on the web page, without applying proper sanitization. There are three forms of XSS:

- **Reflected XSS**: reflected (or non-persistent) XSS happens when user input received by the web application is used directly in the response rendered by the application. It is a server-side vulnerability, since the web application code running on the web server, should have applied proper input validation or sanitization before displaying the user input. What an attacker typically does is sending the victim an URL which contains the client-side code to be injected as part of the GET query string. When the victim opens this URL, the XSS code is send to the application and then reflected back to the victim, who believes that it is a legitimate part of the response from the application. If the response contains JavaScript code, then the victim’s browser will execute it.

- **Persistent XSS**: persistent (or stored) XSS is similar to the non-persistent type, however in this case the XSS is not immediately reflected back by the application, but is stored on the server and then displayed on a web page every time somebody visits it. A typical attack scenario is that a user is allowed to enter some text for his profile, in which he enters the XSS code, storing it on the server. Every time somebody visits his profile, the profile with the XSS is loaded from the database and then displayed to the victim. Persistent XSS is a more serious threat than the non-persistent variant, since it can target a much larger number of people at the same time and is harder to detect by the victim, although the web application’s administrator can probably detect it more easily.

- **DOM-based XSS**: DOM-based (or client-side) XSS is more different then the other two types. In this type of XSS, the vulnerability lies not in the server-side code, but in the client-side JavaScript code of the application. In fact the server does not even see the XSS code. DOM-based XSS occurs when the client-side code, uses some form of user input, for example a piece of the entered URL, to generate a part of the web page. DOM-based XSS is undetectable by the server, but it can be detected by the victim. It is harder to perform for an attacker as well, since he must target individual people.

**Detection**

The most obvious way to find XSS vulnerabilities is to try to input special client-side code in every available input and test if the injected code is present in the received output. A common way is to try to insert the HTML `<script>` tag somewhere, since then it is also possible to insert custom JavaScript. This testing can also be automated. However, there are some special circumstances to take into account:

- XSS vulnerabilities are context-dependent. It cannot only be performed by injecting a `<script>` tag, but also in many other ways, for example by injecting something in an already existing script tag or in an event attribute (such as the `onclick` attribute) of existing HTML elements. The browser will also execute this as JavaScript, allowing it to be misused for an attack. It is even possible to perform XSS in HTML comments, simply by ending the comment. Some constructs can be perfectly safe to use in one context, while in another they could be dangerous.

- In some cases, XSS can also be performed through Cascading Style Sheets (CSS). This form of XSS is also called CSS injection. Some browsers, execute JavaScript present in certain special CSS constructs. CSS is generally considered safe, since it is only used for styling, and therefore protection against CSS injection is easily forgotten. Probably, the most well known example of CSS injection is the MySpace Samy Worm\(^\text{24}\).

- With persistent XSS, the location of injection and the location where the injected content is displayed could be different. For example, when users can enter some text for their profiles, the location of injection is the section in the user settings page where this text is entered, while the text is being shown on the profile page, which is visible to other users. This separation makes it much harder to (automatically) detect XSS vulnerabilities.

\(^{24}\)In 2005 the hacker Samy build a “Worm” for MySpace using CSS injection. With this worm he obtained over one million friend requests in less than twenty hours. For the complete story and technical explanation see: [http://namb.la/popular/](http://namb.la/popular/)
With DOM-Based XSS, the injection takes place through some user controllable JavaScript variable, such as `document.location` which is controlled by the URL. There is no clear input, such as a form field, that could be tried. Therefore, it might not always be clear where the injection should take place to perform DOM-based XSS, which makes it harder to detect this kind of XSS. Other detection techniques should be employed, such as searching in the JavaScript source to check if it uses known user controlled variables in constructs that could possibly cause XSS, such as JavaScript’s `document.write` function.

Protection

There are many different ways to protect against XSS. Some techniques are not even implemented in the web application, but rather in the client’s web browser. However, these protection mechanisms are not very common and sanitization by the web application remains the industry standard [WSA+11].

Web applications usually prevent XSS by using a blacklist approach to remove all HTML tags before displaying user content on the web page. Even better is to escape all special HTML characters by translating them to their safe HTML entities. Using a whitelist approach is also possible, if there are some constraints on the value to display, for example when you know it should be a number. If however the user must be able to enter completely free text, then using a whitelist approach is not feasible. It becomes even more interesting when the user is allowed to use some safe HTML tags for mark up, such as the `<strong>` tag (for bold text), while unsafe tags are prohibited. This could be achieved by searching for all HTML tags, and then removing the ones prohibited (blacklist in blacklist approach) or keeping the ones allowed (whitelist in blacklist approach). A whitelist approach is in this case much safer, because there are many ways to inject JavaScript and there could be even more in the future.

Protecting against XSS is not straightforward. The method described above, removing all or some HTML tags, will only work against XSS caused by injecting HTML tags. It will not work against XSS in event attributes of HTML tags or in JavaScript itself. It will also not work against CSS injection. For these kinds of XSS, other sanitize functions should be used. The approach to filter all special HTML characters will probably work in many cases as well, but not always. Therefore, special attention should always be applied and the specific use-case and context should be well considered to determine the protection strategy.

Web frameworks usually provide functions for sanitizing the user input. This could be either an HTML tags removal (strip) function, a special HTML characters escape function, or most commonly both. Furthermore, they can provide special functions for XSS prevention in different contexts, such as a sanitize function for the CSS context. In some frameworks the strip tags function takes a whitelist or blacklist specifying what tags are respectively safe or unsafe, to provide the developer more control over what should be removed and what not. As for DOM-Based XSS, the framework could supply a JavaScript library with functions to escape special HTML characters, although this is not common.

The protection could be automated by integrating it in the framework’s template engine. When the template engine displays a piece of data, such as some user input variable, it could automatically apply a sanitization helper. However, as discussed before, XSS is context dependent and good protection should be aware of that [WSA+11]. In theory it is possible to make the automatic protection context-dependent, but in practice this is not very common. Therefore, developers should not completely trust on the automatic protection.

Besides ways to directly prevent XSS, some web frameworks also offer some indirect protection mechanisms. These mechanisms focus on reducing the impact of XSS attacks. For example, using XSS it is possible to hijack another user’s session. However, some web frameworks offer the possibility to bind a user’s session to his IP-address, making it harder to steal his session. Another example is the encryption of cookie data, making the leakage of confidential information in someone’s cookies less likely.

2.2.4 Cross-Site Request Forgery protection

Vulnerability

Cross-Site Request Forgery (CSRF) is an attack where an attacker tricks some user, or his browser, into loading a request to perform some action on a web application on which that user is currently authenticated [ZFP08]. The web application then believes this request was performed willingly by that user.
Chapter 2. Web frameworks and security

authenticated user and executes it. For example, an attacker could send an e-mail or post a message on a forum containing the URL “http://www.example.com?action=delete_my_account”. When someone authenticated at example.com clicks on this link, his browser will open that page and example.com will think the user wants to destroy his account. CSRF exploits the trust a site has for the user’s browser. This differs from XSS, which exploits the trust a user or his browser has in a particular site.

To perform a CSRF attack, the following conditions must be met:

1. The web application involves HTTP requests with side effects, i.e. HTTP requests that cause actions to be performed.
2. The application relies on a user’s identity that is automatically send with each request (by the browser), after a possible initial authentication step.
3. The attacker has a way to trick a user or his browser into performing an HTTP request.

All are very common and easily achieved. Many websites use sessions with session IDs to authenticate the user after the initial login. As for the last condition, there are many options, such as sending an URL to the user by e-mail, or post it somewhere on another website. The attacker could even perform a “stealth” CSRF attack, by embedding the URL inside an image tag. The user’s browser will then try to load the image, thus performing the request, without the user noticing.

Detection

CSRF is usually detected by scanning web applications for the first two conditions. If it has forms or links triggering an HTTP request with side-effects and uses some identifier that is automatically send with each request, then the application is potentially vulnerable to CSRF, if no countermeasures are employed.

Protection

There are various ways to protect against CSRF. The most obvious is to somehow verify that the request was performed by the actual user, from within the web application. This can be done by checking the HTTP Referer or Origin header, which contains the point of origin of the request. If the request did not come from within the web application itself, then it should be ignored. This approach is commonly used by web frameworks and can be employed automatically. However, unfortunately, it is not waterproof. In some cases it is possible to spoof HTTP headers. It is also possible that the attacker can distribute the URL with the CSRF attack within the application, for example by putting it in his user profile on that application.

A better approach is to use a secret, user/session-specific token in all requests with side effects. In forms it can be included as hidden field and in URLs as extra (GET) parameter in the query string. The web application should remember the token, check it for each applicable request and ignore requests with wrong tokens. This prevents CSRF, because the attacker does not know and cannot guess the token and thus cannot construct a valid request. Many web frameworks offer this approach and provide helpers to generate and verify these CSRF tokens. In this case, the developer must include the generated CSRF token in all applicable forms and URLs. However, the protection can also be applied automatically, in which case it is integrated into the frameworks form builder and helper functions to construct links. The verification of the token is then automatically performed for all requests, and the only thing the developer should do is use the provided form/link builders instead of constructing his own.

Besides the two protection mechanisms explained above, there are also some other, less common protections that frameworks could apply. They could for example try to reduce the attack window, by shorting the session life time. Another approach is to let the user perform some non-automation check (e.g. with a CAPTCHA) to verify it is really a user performing the action, or let him re-authenticate (e.g. enter his password again). However, these approaches are very cumbersome and annoying.

2.2.5 Protection against HTTP Header injection

Vulnerability

HTTP Header injection is a vulnerability that can occur when the web application includes user input in the HTTP response headers [SP07]. If this is the case, an attacker might be able to insert his own headers,
which can lead to false redirection, XSS or HTTP response splitting. An example of a vulnerable web application would be an application that uses the page parameter from the GET query string to determine the location for a redirect. The page parameter ends up in the location response header as Location: http://safe.com/<page>. However, if the attacker crafts an URL with example.html%0d%0aLocation: http://evil.com as page parameter, then the application could redirect to the evil.com domain. This is because %0d%0a is the URL encoding for \r\n, Carriage-Return and Line-Feed (CRLF), effectively creating an extra HTTP response header.

A special kind of header injection attack is HTTP Response Splitting [SP07]. With this attack there are two CRLFs injected, instead of just one. By the HTTP standard, headers are separated from each other by one CRLF and the headers are separated from the body of the response with two CRLFs. Thus, by injecting two CRLFs, the attacker can take control over the body and insert content at his will. This could allow the attacker to display false content on the web pages, or enable him to perform XSS attacks.

**Detection**

Header injection attacks can be found by trying inputs with a CRLF and a custom header and then inspecting the response for this particular custom header. The same goes for HTTP Response Splitting. You simply try to enter two CRLFs, followed by some custom content in every available input and determine if the body of the response contains this custom content. These detection techniques could also be easily automated.

**Protection**

Web applications can prevent these kind of attacks by only allowing certain values for parameters in response headers (whitelist). They could also use a blacklist approach, by removing or escaping the special CRLF character. Web frameworks can help with this, by supplying a helper to perform this task. Some web frameworks also supply helper functions to set values for certain common HTTP headers. For example, redirects are very common in web applications, thus many frameworks supply a redirect function that sets the Location header for you. In this case, the framework could apply the sanitization on the given input values automatically.

### 2.2.6 Protection against Unvalidated Redirects

**Vulnerability**

An unvalidated redirect occurs when the web application performs a redirect to a location supplied by user input [Pro13]. This could allow an attacker to hide malicious links in URLs of trusted websites. For example, there is a web application, running on safe.com, that takes a single query string parameter url and redirects to it. Then an attacker could craft a malicious link http://safe.com?url=evil.com and send it to his victim. When the victim opens this link, he will redirect to evil.com, but think it is part of safe.com, since the URL started with the trusted safe.com domain. An attacker could use this to lure people to sites containing malware or password phishing sites.

Unvalidated redirects differ from HTTP header injections, since there are no new headers injected. Instead, a value is set for the normal redirect header and the problem is that this value is not properly validated and can point to an external location. Furthermore, HTTP header injection can occur for all response headers, whereas unvalidated redirects only for some. When header injections are prevented, unvalidated redirects could still occur and vice-versa.

Unvalidated redirects are not very harmful, but still take number ten in the OWASP top 10 [Pro13].

**Detection**

They can be found easily by trying to enter an URL to another site in every available input and see if the web application tries to redirect to that other site. If so, then an unvalidated redirect is found.

**Protection**

Web applications can protect against these attacks by simply not use redirects at all, or, somewhat less extreme, do not use user input in URLs any more. In many cases the latter is possible. When it is not,
then the application should validate the supplied user input before using it in the redirect. The best way to do this is by using a whitelist of allowed values. In case the redirect should always be local, i.e. the redirect’s location is on the same domain the application is running on, then a blacklist approach could also work. The blacklist could for example remove unsafe constructs that point to an external location, such as http://. Even better is to parse the URL and extract the path, ignoring all the host information, so the redirect will always be local and thus safe.

Web frameworks could help with this by providing sanitization helper functions or by providing methods to extract the path from a location. The protection could also be applied automatically, when the framework supplies a redirect helper as discussed earlier for HTTP Header injections. The drawback of this approach could be that the web application itself also cannot redirect to external URLs any more, but this could be solved by creating an optional second argument that indicates that external URLs are allowed in this case.

### 2.2.7 File Inclusion protection

**Vulnerability**

A file inclusion vulnerability allows the attacker to change the reference to some file to another file, such that the web application performs some action on this new file. This is possible if the application uses user input to establish the location reference of the file. Such an attack could allow the attacker to access sensitive data, destroy important data or even execute arbitrary code, depending on the action the web application wants to perform on the file. There are two flavours of File Inclusion attacks [SP07]:

- **Local File Inclusion (LFI)**: the attacker performs a file inclusion attack involving a local file on the application’s server. LFI is usually used to obtain sensitive data or to overwrite important data, possibly to make the application or server malfunction (Denial of Service).

- **Remote File Inclusion (RFI)**: the attacker performs a file inclusion attack involving a remote file, i.e. a file on some other server, that is accessible through the internet. RFI is usually used to inject and execute arbitrary code.

An example of an LFI attack would be a web application that allows a user to read any file in a certain folder by giving the filename. The application uses the entered filename to open the file as follows: `open(/path/to/folder/<file>)`. One would think the user can only access files in the given folder. However, an attacker could break out of the path to `folder` by using relative paths and a technique called Path Traversal, and access any file he wants. For example if he enters `../../etc/passwd` as filename, then the location would be `/path/to/folder/../../etc/passwd`, which translates to `/etc/passwd`, allowing the attacker to read the secret passwd file (on UNIX systems). Overwriting files is done similarly, only this time the application wants the user to provide a filename to write some content to. The attacker could use this to overwrite the passwd file, which will likely cause the server to malfunction.

An RFI attack is not used to read or overwrite sensitive data, but to load and execute code. Consider for example an application that dynamically includes the module of which the name is supplied by the user. The application uses the following code to do so: `include(<module_name>.mod)`. If the attacker now enters `http://evil.com/code`, then this would cause the application to include the `code.mod` file on `evil.com`. Since the attacker controls this file, he can include any module he wants and use it to execute arbitrary code in the web application’s environment.

**Detection**

File inclusion vulnerabilities are usually detected by trying to enter locations of files and check the response for presence of that file. For example, a local file inclusion could be detected by trying `/etc/passwd` as input. On UNIX servers, the response will contain something like `root:x:0:0` if the application is vulnerable. Remote file inclusions are detected in a similar way. The attacker tries to input a URL to an earlier created file that contains code (written in the same language as the application) that prints some string, such that it should end up in the response. The attacker can then check the response for this particular string.
Protection

The best way to prevent file inclusion attacks is to check all inputs used in file references against a whitelist. If this is not feasible or cumbersome, then it could be an option to remove all special characters for file access, such as the forward slash (/), backslash (\), dot (.) and NULL byte (%00)\textsuperscript{26}. It is also possible to filter special constructs such as http:// or :// to prevent RFI. These blacklist approaches are helpful, but not foolproof. For example, special attention should be paid to various encoding options an attacker could use to masquerade his attack. For these reasons, a whitelist approach is always recommended.

Web frameworks could offer functions to help protect against file inclusion attacks. They could for example supply a function to check if a filename is safe, i.e. only contains safe characters, such as alphanumeric characters. This is a safe whitelist approach. They could also offer helper functions to sanitize filenames, i.e. remove all unsafe characters. The framework could apply this protection automatically. In order to do so, it should supply a function to open files, which takes two arguments: the base path and the filename. The base path is considered safe and should not contain user input. The filename could be unsafe and is therefore sanitized.

2.2.8 Mass assignment protection

Mass assignment or over-posting\textsuperscript{27} is a security vulnerability found in some web frameworks, among which ASP.net MVC and Ruby on Rails\textsuperscript{[McK11]}. It is not as common as some of the other vulnerabilities, but since it is a serious threat for Ruby on Rails application, we have chosen to discuss it as well.

Vulnerability

Mass assignment is a way to quickly assign new values for a model’s attributes, by proving a hash (associative array or object) with these values. This is often used to update a model’s attributes with user supplied values through a form. The hash contains the form values, having a key-value pair for each of the model’s attribute. The hash is mass assigned to the model, meaning the model’s attributes will get the value of the corresponding item in the hash. While mass-assignment allows a developer to quickly generate model creation and update forms, it could allow users to set values for restricted attributes. This is as easy as simply adding an extra key-value pair to the hash, which in a form could be done by adding an extra input field and providing it with some value.

An example of an attack would be an application that has a user model, with three attributes: username, password and admin. The admin field is restricted and defaults to false, while the username and password are user supplied. Users can register for the application by filling in the registration form, which has fields for the username and password fields. However, a user could now create an extra field with the name “admin” and set it to true. If mass assignment is used, then the newly created user will be an admin.

Detection

Mass assignment vulnerabilities can be detected by trying to add extra fields for attributes that normally should not be editable. If the values for these attributes can be changed this way, then the application is vulnerable.

Protection

Web frameworks often supply a method to specify which attributes are safe for mass assignment and which are not. The mass assignment functionality will take this into account and only update the safe attributes, while values for other attributes are completely ignored. This protection is often applied automatically, as long as the developer somehow specifies the safe attributes. If this is not specified, then mass assignment will either not function at all, which effectively prevents the vulnerability, or the framework allows every attribute to be mass assigned, meaning the vulnerability still exists. The specification can either be whitelist based, by specifying which attributes are safe to mass assign, or blacklist based by specifying which attributes should not be mass-assigned. The preferred method is a whitelist approach, to make sure some restricted attribute is not forgotten. The specification is either

\textsuperscript{26}See https://www.owasp.org/index.php/Path_Traversal for a complete overview.
done at model-level, specifying which attributes can be mass assigned for a particular model, or at
controller-level, specifying which attributes can be mass assigned for the models updated by that specific
controller. The latter approach is more flexible, making it for example possible to differentiate between
user controllers and admin controllers.

2.2.9 Session management

Since sessions often store authentication data, such as the user that is currently logged in, they should be
managed properly [Pro13]. Especially the session identifier, that tells the web application what session
belongs to which user, should be handled with care. If an attacker obtains another user’s session identifier,
then he can impersonate that user. Some vulnerabilities related with session management are:

- **Session Fixation**: with session fixation an attacker tries to preset the victim’s session identifier,
  so that he knows it. Frameworks could prevent this by offering a way to reset the session identifier.
  Frameworks that include an authentication framework could also implement this automatically by
  resetting the session after successful authentication.

- **Session Prediction**: the session identifier is predictable because it depends on publicly known
data and/or uses a weak generation algorithm. This allows an attacker to guess the session identifier
of another user and thus steal his session. Since frameworks often handle session management, they
  can prevent this by using a (pseudo-)random session identifier instead of, for example the username,
  which is predictable.

- **Session Hijacking**: an attacker somehow steals another user’s session identifier. This is for ex-
  ample possible by sniffing the network connection. Frameworks could prevent this by only allowing
  HTTPS connections, which are secure. Also XSS attacks could allow an attacker to obtain a user’s
  session identifier. Frameworks could of course help to protect against XSS too, as discussed in
  section 2.2.3.

2.2.10 Other security functionalities

Frameworks could also offer security functionalities that do not directly protect against a specific security
vulnerability, but are however security related. Some examples are:

- **Input validation**: frameworks could offer means to automatically validate user input, such as
  values entered in form fields. Although this does not directly fix a security vulnerability, it could
  help prevent them. It will be especially helpful against injection attacks.

- **Form tampering protection**: prevention of tampering with ‘locked’ form values, such as values
  in hidden input fields or in drop-down fields. Usually this method involves a digital signature or
  message authentication code, for example with an HMAC [BCK96]. This way the application can
  trust the data entered in these fields and there is no need to retrieve it from some secure channel
  (e.g. the database) again.

- **Secure cookie storage**: prevents tampering with data in cookies, because all data is authenticated
  (usually with an HMAC). This makes it unnecessary to always use a secure back-end storage device,
  such as a database. Secure cookie storage could also be used to store session data, as is done by
  Rails by default, although the developer should be aware of replay attacks.

- **Secure file upload**: makes sure uploaded files are locked in a directory (not possible to overwrite
  system files), cannot be executed and are checked for malware. This could help to prevent or reduce
  several attacks.

- **Anti-Automation**: prevents forms from being automatically filled-in, for example by using CAPTCHAs
  [ABHL03]. This helps to stop spam and brute-forcing of passwords and accounts.
Chapter 3

Ruby on Rails

As mentioned in the introduction, we have chosen Ruby on Rails as first test case for our approach to analyse the security of web frameworks. We have chosen for Rails for several reasons. First of all, Rails is a popular framework. It is used by over more than five hundred thousand websites and with that it is the most popular web framework for Ruby. Secondly, and more importantly, there were relatively many vulnerabilities found in Rails recently. Some of these vulnerabilities even had disastrous consequences, such as the CVE-2013-0156 vulnerability mentioned in the introduction. This vulnerability threatened a huge number of websites at once, among which DigiD, the centralized authentication solution of the Dutch government. The increase in amount and severity of vulnerabilities in Rails, was one of the main motivations to start this research, so it seems rather logical to also use Rails during the rest of this research as well.

The rest of this chapter introduces the Ruby language and the Rails framework. Furthermore it presents important security features provided by Rails and list some of the security vulnerabilities Rails had to deal with in the past.

3.1 Ruby and Rails

Ruby on Rails is a web framework that runs on the Ruby language, which is a dynamic, interpreted, object-oriented language created by Yukihiro “Matz” Matsumoto in 1995. It is inspired by languages as Perl, Smalltalk, Eiffel, Ada and Lisp and is meant to balance functional programming with imperative programming. Ruby is fully object-oriented, meaning everything is an object and there are no primitive types. Except for the basic types such as strings, integers, et cetera, it also has some special types, such as symbols (named constants starting with a semicolon), arrays (lists), hashes (associative or named arrays) and ranges (a range from an object to another object). The language features single inheritance only, but it has the concept of modules, which are collections of methods that can be included in any class. Another key point of Ruby is that it is a very flexible language. It allows users to alter its parts, by removing or redefining methods and classes at will. The concept of blocks also illustrates this. Blocks are closures which can be used with many methods to fill in the details of what such method should do. The most recent version of Ruby is 2.0.0, which was released February 24, 2013. However, Ruby 1.9.3 (released at October 31, 2011) and even Ruby 1.8.7 (released at June 1, 2008) are still widely used.

Ruby is shipped with a package manager, RubyGems, which contains gems: libraries, plug-ins and even full Ruby programs. Ruby on Rails (or Rails for short) is one of these gems. It is a web application framework that was created by David Heinemeier Hansson in 2004. Ruby on Rails combines a push-based Model-View-Controller architecture with RESTful (Representational State Transfer) routes [Gee06, BK07, Fie00] and is especially suitable for applications performing many Create, Read, Update and Destroy (CRUD) database actions. These applications can be developed very quickly thanks to common software engineering patterns and principles such as the Active Record Pattern [Fow02], Convention over Configuration and Don’t Repeat Yourself (DRY programming) [Gee06, BK07]. However, because of Rails’ modular design and the power to override and extend almost everything, partially thanks to the Ruby language, developing non-standard CRUD applications is also possible.

[3] For more information about blocks or Ruby in general, see http://www.ruby-lang.org/en/about/
Like many other frameworks, Rails is based heavily on the Model-View-Controller architecture. The models are written completely in Ruby code and represent data objects, that are usually stored in a database. The interfacing between the database and the models is performed largely automatic thanks to the Active Record pattern. The views in rails show the data objects and usually consist of HTML in combination with ERB (Embedded Ruby). However, views can also contain other formats, such as XML or JSON. Finally, there are the controllers, which form the interface between the models and the views. These are also written in pure Ruby. Besides the three standard MVC components, Rails also uses a routing system. This routing system is used to translate web addresses to the appropriate controller actions and vice-versa.

The framework itself has a modular design. It consists of six main components\footnote{Earlier versions of Rails consisted of seven main components. The extra component, Active Resource, provided a mapping between business objects and RESTful web services. With it, it was possible to use web-based resources as local objects with CRUD semantics. It was extracted out of the Rails core with the 4.0 release.} which are all separate gems and thus could be installed separately if wanted\footnote{See \url{http://guides.rubyonrails.org/getting_started.html#the-components-of-rails} and \cite{BK07} for more information on Rails’ components.}

1. **Action Mailer**: a framework for building e-mail services.

2. **Action Pack**: the action pack takes care of the Controller and the View part of Rails. It consists of three subparts:
   a) **Action Controller**: the controller part of MVC. It forms the base for all the controllers. It processes incoming requests and outgoing responses and provides functionality such as session management.
   b) **Action View**: the view part of MVC. Action View is Rails’ template engine. It handles the rendering of templates and includes a form builder.
   c) **Action Dispatch**: the base for Rails routing system. Rails is heavily based on the RESTful (Representational State Transfer) architecture \cite{Pie00}, which is implemented in Action Dispatch.

3. **Active Model**: the model part of MVC. It is the base of all models in Rails. It is also the interface between Action Pack and Active Record. Active model provides services such as model validations.

4. **Active Record**: Rails’ default ORM. It extends Active Model and forms the interface between Rails’ models and the database of choice. It provides standard CRUD functionality, methods for finding records, et cetera. Currently Active Record supports MySQL, PostgreSQL and SQLite out of the box, but it can be extended with others (such as SQL Server, Sybase and Oracle) using gems.

5. **Active Support**: a collection of utility classes and standard Ruby library extensions. It is used heavily by Rails’ core code, but can also be used in the user’s applications.

6. **Railties**: the core of Rails, which provides code to build new applications and is the link between the various components of Rails.

### 3.2 Rails versions

Rails versions are represented by three integers: the major version, the minor version and the build version. When the build version changes (the last integer), it means some implementation detail changes that does not break anything, i.e. it is fully backwards compatible. This happens for bug fixes, security fixes and performance improvements. When the minor version changes (the second integer), it means the version may add new features but is still backwards compatible, except for some deprecation warnings. When the major version changes (the first integer), it means it may change the public interface (API) of the framework and could break software written in older versions. In other words, the change is not backwards compatible.

At the time of writing the most recent version of Rails is 4.0.0, which was released June 25, 2013. However, there are still many applications running on older versions such as 3.2.x, 3.1.x and even 2.3.x,
3.3 Security functionalities of Rails

Ruby on Rails offers a large amount of security functionalities which can help to protect against common attacks7. We will now list the various security features and protection mechanisms offered by Rails and explain how these are implemented.

SQLi protection in Rails

Protection against SQL injections in Rails is handled by Active Record. This module supplies a complete and partially automatic database interface with sanitization helpers and query builders. Rails implements a blacklist approach using the `sanitize_sql` helper, which will escape the special SQL characters: ', *, \n (line breaks) and NULL characters. This helper method is used internally in Rails, but also available for developers building a web application in Rails.

When using the query builders or the ‘automatic’ features of Active Record, then the protection is largely applied automatically. In Rails, it is almost never necessary to write an SQL query yourself. Sometimes the query is generated and executed automatically. For example, to save a model to the database, Rails supplies the `save` method, which will automatically generate the corresponding INSERT or UPDATE query and executes it. This save method, will also apply sanitization on all values of the model’s attributes. When performing more manual actions, such as retrieving (selecting) objects from the database, the developer can use Rails’ ‘query builders’. For example the call “Book.where(‘author = ?', ’Someone’).limit(10).all” translates to “SELECT * FROM books WHERE author = ’Someone’ LIMIT 10”. In this case the value for author (’Someone’) and the limit (10), are sanitized automatically.

However, the methods should be used correctly and not all of Rails’s query builder methods are automatically protected. For example, the query above could also be performed with “Book.where(‘author = #{Someone}’).limit(10).all”, in which case the value for author is inserted (using string interpolation) in the query directly without applying sanitization. An example of a query builder method not automatically safe is order, which supplies an ORDER clause. This method takes a column name and does not supply any sanitization at all. The reason for this, is that in SQL the column name in the ORDER clause is not bind by quotes, thus a blacklist approach will be insufficient. Therefore, to securely use the order method, the developer needs to apply his own protection using a whitelist approach on the supplied column name.

Command Line injection protection in Rails

The Ruby language (and thus Rails) offers a simple protection against command-line injection, by providing a method `system` to execute system calls, that takes a dynamic number of arguments. The first arguments is the command, which is considered safe and thus not escaped. All the other arguments are considered to be arguments for the command, which are bound by quotes and properly escaped. Thus, when using this method in the right way, protection against command line injection is automatically applied.

XSS protection in Rails

In Rails, XSS protection is implemented in the Action View and Active Support (in ERB::Util) modules. The implementation is somewhat similar to the implementation of the SQLi protection. Again, the framework uses a blacklist approach, and supplies sanitization helpers and “builders” that can be used to build common HTML constructs, such as links and forms. The most important helper is the `html_escape` method.

---

7https://groups.google.com/forum/?fromgroups=#!topic/rubyonrails-security/G4TTUDDYbNA
http://guides.rubyonrails.org/security.html
method, which escapes all dangerous HTML characters (&, >, <, ", ') to their safe HTML entities. However, Rails also supplies other helpers for other contexts or to allow selective filtering. Important other helpers are:

1. ERB::Util html_escape_once: same as html_escape, but ignores html_safe status (more on this later).
2. ERB::Util json_escape: escapes HTML entities (&, >, <) in JSON strings to Unicode escape sequences and removes double quotes (*). 
3. ActionView escape_once: calls ERB::util escape_once.
4. ActionView strip_tags: removes all HTML tags from a string using a blacklist approach (stripping by regular expression). It leaves the content within the stripped tags. Also strips comments.
5. ActionView sanitize: removes HTML tags and attributes that are not included in the given whitelist. This method keeps the content within the tags it filters, except for the script tag which is filtered out completely. It searches for tags and attributes using regular expressions. When no whitelist parameter is given, it uses a default value, that only removes dangerous tags and attributes, such as the script tag or the onclick attribute.
6. ActionView sanitize_css: sanitizes a block of CSS code using regular expressions and whitelist of safe CSS properties. This method is also used by the sanitize method when it comes across an style attribute.
7. ActionView escape_javascript: escapes strings for use in JavaScript (string) segments. This method will escape single ('') and double quotes ("), as well as carriage returns (\n).

Since Rails 3.0 the framework applies XSS protection automatically. This is implemented by automatically calling html_escape whenever there is a variable outputted in an ERB template (using the <%= variable %><% = variable% > construct). It is possible to circumvent this, by marking the string as safe using the html_safe method or the raw method. This could be used to allow for more selective filtering, using the sanitize method for example. Rails does not apply context-sensitive sanitization, but simply uses the html_escape method for all contexts, which in most cases will suffice, but could be insecure in some circumstances as discussed in [WSA+11].

As said before, Rails also supplies several builder methods to build common HTML constructs. For example Rails offers a content_tag method to create a general HTML tag with content. The output of these builders is considered HTML safe and will not be sanitized. The input of these methods however is automatically sanitized. So the content and attributes supplied to the content_tag method, will be automatically escaped by Rails. The same holds for the other builder methods. These include the tag method and methods to build links and buttons (button_to, link_to, link_to_if, ...) and forms and form fields (form_tag, input_tag, submit_tag, ...).

CSRF protection in Rails

Ruby on Rails also offers protection against CSRF attacks. This protection is applied fully automatic. The only thing the developer needs to do is use GET and non-GET requests appropriately, as is also required by W3C. This means that GET requests should be used when the request is more like a question (only retrieves data) and does not have side-effects. In other cases, when the request is more like an order and does have side effects, a non-GET request should be used, for example a POST or PUT request. This already helps a bit to prevent CSRF, since automatically sending GET requests is much easier than automatically sending non-GET requests. However, this alone is not enough [ZFO8].

Therefore, Rails uses a special security token to protect non-GET requests from CSRF as well. This security token, also called CSRF token, is calculated from the current session identifier and a server-side secret. It is automatically included in all forms and AJAX requests generated by Rails. Rails will automatically verify the token for each request. If the send token does not match the expected value, then the session is reset, effectively logging out the user. This system helps to prevent CSRF, since an attacker does not know the CSRF token and thus cannot construct a valid request performing some

---

[WSA+11]: https://www.w3.org/2011/CR-cors/
[ZFO8]: http://www.zapier.com/blog/2008/10/prevent-csrf-attacks/
action. However, this system will only work when the security token stays secret. Therefore, it will fail in case of XSS vulnerabilities, since these can be used to extract the token from the web page.

This protection mechanism is implemented in the Action Controller component (the generation and verification of the token) and the Action View component (the automatic inclusion of the token). It is applied fully automatic, as long as the developer clearly distinguishes GET and non-GET requests and uses Rails helpers to build forms, links, AJAX requests, et cetera.

**Header injection protection in Rails**

Ruby on Rails does not offer full header injection and response splitting attacks protection. It does however, offer a redirect helper (redirect_to) (implemented in Action Controller), which will automatically escape the provided URL to prevent these attacks. So when this method is used to set the location header, then header injection and response splitting attacks are automatically prevented. However, for other response headers manual protection has to be applied.

**Protection against Unvalidated Redirects in Rails**

Rails’ redirect_to helper method can also help to protect against unvalidated redirects. The method can take a hash in which case the URL to redirect to is build based on the values in this hash. If in this hash the value for key :only_path is set to true, then the framework will make sure the resulting URL contains only a path and thus is local. The method can also take a string as argument, in which case the protection should be applied manually. Rails does supply a helper to do so, namely by parsing the URL and extracting the path with \texttt{URL.parse(url).path}.

**Secure cookie storage in Rails**

The default storage method for session values in Rails is cookie store. This means session data is stored in a cookie locally on the user’s computer. The information inside it is however protected using a secure cookie storage mechanism, implemented in the Action Controller and Action Dispatch. Rails authenticates the data inside the cookie, but does not encrypt it. So, the session should not be used to store any secrets. To verify the data, Rails calculates a hash (using SHA512) of the data in combination with a server-side secret called \texttt{secret_key_base}. The user thus cannot tamper with the data, because then the hash will not match any more, which is detected by Rails.

There is one attack Rails developers should be aware of when using sessions, namely the replay attack. Although user cannot modify the session data at will, they can however replay it by replacing a session cookie with some older version. Thus the session should not be used to store things like credits, because then the user could reset his credits at will. A common approach to solve this problem is to store this kind of data in the database, instead of in the session. This will also have the benefit that credits are persisted between sessions, i.e. the user cannot reset his credits by logging out and logging in again.

Another solution is to not use cookie storage for session values, but some other method, such as database storage. This will solve the replay attack problem, but will lower performance and requires somewhat more time to setup.

**Session management in Rails**

Ruby on Rails also provides some protected measures against common session management attacks, which are implemented in the Action Controller and Action Dispatch modules. First of all, Rails uses an unpredictable 32 byte long MD5 hash value as session identifier, to counter prediction attacks. Secondly, Rails provides a \texttt{reset_session} helper to block session fixation attacks. This protection is not enabled automatically, since Rails by default does not provide an authentication framework. However, many Rails developers do not write their authentication functionality themselves, but use a gem (library) such as Devise. Such libraries (almost always) do apply the reset_session protection automatically. Finally, Rails helps to prevent session hijacking by providing a configuration option to force secure SSL connections and of course by providing several XSS protection mechanisms.
Mass assignment protection in Rails

Ruby on Rails is vulnerable to mass assignment attacks. Fortunately Rails supplies a protection mechanism. Before, in Rails 3 and earlier, this was a model-based approach, implemented in Active Model. Developers could either use the `attr_accessible` helper to specify the safe/allowed attributes (whitelist) or the `attr_protected` helper to specify restricted/disallowed attributes (blacklist). The framework uses the supplied specification to automatically filter the parameters supplied to mass assignment update methods, such as `update_attributes`.

In Rails 4, the mass assignment protection is replaced with Strong Parameters. This is a controller-based approach, implemented in Action Controller. It allows to specify a list of permitted parameters for a particular model inside the controller. The protection mechanism will automatically filter out all other parameters. With strong parameters, the developer has much more flexibility. He could allow one controller to update only some attributes, while another (admin) controller could be allowed to update anything. Furthermore, developers cannot only specify which attributes are allowed, but also the type of the attribute allowed: scalar value, array or hash. This allows more fine-grained control when using nested models, a feature of Rails to update multiple models easily with a single form.

3.4 Recent vulnerabilities in Rails

Since Ruby on Rails was first introduced, there are a significant amount of vulnerabilities found in it. This section will discuss these vulnerabilities and summarize them into statistics. We use this to determine what we should include in the benchmark, which is presented later in section 4.3.4.

The vulnerabilities found in Ruby on Rails are announced in a Google group (mailing list)\(^9\). This also forms a nice back-log of all the vulnerabilities found in Rails, or at least the ones found after October 12, 2007 (or after Rails 1.2.4). According to this list, there were 61 security announcements in the past six years (from October 12th, 2007 to October 1st, 2013). However, when looking more closely, it appears that some announcements contain multiple vulnerabilities, provide extra information for an earlier announcement, are not directly Ruby on Rails related, or contain best practices instead of real vulnerabilities. When taken this into account, the total number of actual vulnerabilities changes from 61 to 43.

Since this thesis focuses on Rails 4, we decided to only take the more recent vulnerabilities into account. We have chosen to take a closer look at the vulnerabilities in the last three years (found in 2011, 2012 or 2013 up to October 1st, 2013). The vulnerabilities found in this period can be categorized in eight categories, that more or less correspond to the the categories of the common security functionalities in web frameworks back in section 2.2. The rest of this section presents each category and shortly discusses the vulnerabilities found in Rails belonging to them. It concludes with graphs summarizing the found vulnerabilities and the trends that can be derived from these graphs.

3.4.1 Categories

SQL injections

SQL injections in Rails are almost always located in Rails’ Active Record component. An SQL injection vulnerability in Active Record means that there is some method, that should be safe for usage with untrusted user input, while it is not (i.e. it can lead to an SQL injection). This can happen for three reasons:

1. The sanitization helpers that Active Record are not good enough, i.e. they do not disable all ways to inject SQL or they can be circumvented with specially crated strings.
2. The sanitization helpers are not applied at all, i.e. it is simply forgotten in the code, or they are disabled with some option. This was the cause for two of the vulnerabilities.
3. SQL can be injected due to some special features of Active Record. Active Record has several special features that allow programmers to put in special constructs, such as hashes or arrays to easily perform some actions or set Active Record options that determine parts (set values for clauses) of the generated SQL query. However, user input in Rails is sometimes automatically parsed to such

\(^9\) https://groups.google.com/forum/#!forum/rubyonrails-security
special constructs, again to ease developers. In some special cases this could cause attackers to set these options using specially crafted strings, that would be interpreted by Rails. This could allow attackers to inject some SQL, for example to query other tables, or in one case to inject arbitrary SQL, because the attacker could overwrite the options mentioned earlier. This was the cause for the other three vulnerabilities.

**XSS**

The most common vulnerability in Rails is XSS. XSS vulnerabilities in Rails are mostly located in the Action View or Action Controller components or in the string class extensions in Active Support. As for SQL injections, they are mainly caused because either the used sanitization helpers are not good enough (caused 6 of the vulnerabilities) or the sanitization is wrongfully not applied for some reason (caused 7 of the vulnerabilities).

**CSRF**

There was only one CSRF vulnerability in Rails in the past three years. It was located in the Action Controller and Action Dispatch modules. It was caused by two main reasons. The first being that the protection was not always applied. Before, Rails would not apply the protection for some special requests, such as XMLHttpRequest (XHR) requests. Now it is applied simply for every non-GET request. The second reason was that after a CSRF attack was detected, insufficient measures were taken. Formerly, only an exception was thrown (resulting in a HTTP 500 error page), while now the complete session is reset (user logs out).

**Header injection**

There also was only one header injection vulnerability in Rails. It was present in the Action Controller component and occurred because the helper to set the response content type did not apply sanitization, making it unsafe for usage with untrusted user input.

**Denial of Service**

Denial of Service attacks make the web application unusable for normal users. This can be done in many ways, for example by flooding the application’s server with data, but sometimes also because of a vulnerability in the web framework. In Rails this happened two times, both of which were caused because some user input was (wrongfully) interpreted as a symbol. In Ruby, symbols are objects that are not garbage collected. Therefore, an attacker could crash applications, by sending strings that would be converted into many symbols, causing the server to run out of memory. It was fixed by using the entered strings directly, without converting them to symbols first, since this was not really necessary. One of the vulnerabilities resided in Active Record, the other one in Action Controller.

**Information leakage**

With an information leakage vulnerability, an attacker is able to somehow obtain information he should not be able to get. An example would be that the attacker is able to see the purchase history of another user in a web shop, something that usually is confidential. Another example is that the attacker somehow obtains another user’s (encrypted) password that is usually stored in the database. This kind of vulnerability has many causes. It could for example be caused by another vulnerability, for instance an SQL injection attack could also be used to obtain confidential information. However, it can also be caused for many other reasons. Because of this, there is no general protection mechanism against them and therefore we have not yet discussed these vulnerabilities in this thesis.

Information leakage vulnerabilities are relatively common in Rails. They are usually located in Active Record or Action Dispatch. Three of these vulnerabilities are caused by what Rails calls unsafe query generation. This means that an attacker is not able to inject arbitrary SQL, but could able him to add “IS NULL” to the WHERE clause of the query or eliminate a WHERE clause completely, even if the Rails’ developer checks if the value is nil, using the nil? helper method. The other two vulnerabilities where caused by the way Rails routing system translates URLs to controllers and actions, which on there
turn use the action to render a view. It could allow an attacker to render views he should not have access to. They existed because the methods used to sanitize the corresponding URL parts were not sufficient.

**Mass Assignment**

In the past three years there was only one mass assignment vulnerability in Rails. It occurred only when using the blacklist `attr_protected` solution instead of the whitelist `attr_accessible`. The vulnerability was located in the Active Model component and was caused by the inadequate code to sanitize parameter names.

**Other**

Four vulnerabilities were placed in the other category. These all were vulnerabilities that caused multiple problems, such as information leakage, Denial of Service, code injection, broken access control and authentication bypassing. The very serious CVE-2013-0156 mentioned in the introduction (section 1.1) also falls in this category.

All but one of the vulnerabilities resided in the Active Support component. They were caused by problems with a feature of Rails that allows parsing of XML or JSON parameters. In the code responsible for this, user input could be accidentally interpreted and executed. This allows an attacker to do almost anything he wants, by sending specially crafted parameter strings. The other vulnerability was present in the Active Record component and caused by the Rails feature allowing serialized attributes in models. If users could set values for these attributes, then attackers could use specially crafted strings. Those would be wrongfully interpreted and executed, which could lead to Denial of Service or code injection attacks.

### 3.4.2 Trends

We look at the trends in the found vulnerabilities using two graphs. The first, displayed in figure 3.1, displays the amount of security announcements and actual vulnerabilities for each year from 2007 till 2013. From this graph it is immediately clear that especially the last three years, the amount of vulnerabilities is very high. This was one of the motivations for this research. Although the amount seems to be stable (ten to twelve vulnerabilities per year), it is still way too high and it is also high compared to other popular frameworks. Remarkably, the increase in vulnerabilities in the last three years corresponds with the introduction of Rails 3. Rails 3 and 4 are actually still quite alike, and have much more in common than Rails 2 and 3 have. Up to now (October 1st, 2013) there are no vulnerabilities discovered in Rails 4 yet, but looking at these statistics, it is hard to assume Rails 4 is completely secure.

The second graph, displayed in figure 3.2 counts the found vulnerabilities for each category and year. When looking at the nature of the vulnerabilities, we see XSS vulnerabilities dominate by far. This is also one of the reasons why we have chosen to include XSS in the benchmark, as explained later in section 4.3.4. Especially in 2011 and 2012 XSS was very common. There also were relatively many SQL injections and information leakage vulnerabilities. Other vulnerabilities were only incidental (maximum one per year). However, it appears the nature of the vulnerabilities is shifting. In the past year, the amount of SQL injections, information leakage and especially XSS vulnerabilities, has decreased. Instead, there is a new type of vulnerabilities arising, as discussed in the “other” category.
Figure 3.1: Security announcements in the mailing list and actual vulnerabilities in Rails.

Figure 3.2: Actual security vulnerabilities by category and year.
Chapter 4

A benchmark approach

One of the goals of this research is to find a general methodology to analyse arbitrary web frameworks for security vulnerabilities. As briefly mentioned in the introduction, the idea is to create a benchmark of test applications that can be used to analyse the security of the framework. This chapter first motivates this idea and then explains the approach in more detail. After that, we present the design of the benchmark. We do this by first presenting the global benchmark design, after which we present the specific designs for each module.

4.1 Motivation

Establishing a general method to analyse the security of web frameworks is not straightforward. This comes due to certain challenges when targeting web frameworks instead of web applications, of which some already were briefly mentioned in the introduction. This section has a closer look at these challenges and explains how they led to our approach of using a benchmark.

First of all, we need to consider that web frameworks tend to be very large and complex. For example, the Rails framework consists of over 170,000 lines of code (of which more than 160,000 Ruby code) and the total source code package is almost seventeen megabytes large. It is infeasible to analyse all this code by hand, and therefore it would be nice to find a method that can be (partially) automated. As we will see later, our method can be automated, using dynamic security scanners.

However, not only the size of the frameworks is a difficulty, also the large amount and diversity of potential security vulnerabilities make it challenging. As we have seen in section 2.2 there are many types of vulnerabilities threatening web applications and frameworks and this list is not even exhaustive. When analysing the security of frameworks, all these types of vulnerabilities should be taken into account, which makes our task harder. This is another motivation for automating the process. However, even if we automate it, we should still implement “checks” for all these vulnerabilities. It would be nice if we could skip this step and instead reuse the work of others. This also gives the potential benefit that this work and the tools that resulted from it, are already very sophisticated. They already exist for some time, and thus are likely used widely, are well-tested and are improved for their specific tasks. They are likely better than what we would be able to develop in a short period of time. Fortunately, there are many tools available to analyse web applications for many types of security vulnerabilities. We would want to reuse these tools and apply them on web frameworks as well. Unfortunately, these tools cannot be applied directly to web frameworks, but need a web application instead. We therefore want to create some interface between the frameworks and the tools. This is exactly the purpose of the benchmark. The benchmark consists of test applications that can be analysed using the tools. The results of this analysis could indicate vulnerabilities in the framework.

Another challenge is that a web framework is not a finished product. A framework by itself has no use. It cannot be run on some server and (end-)users cannot use it. In order to turn the framework into something useful, you need to develop a web application in it. The web application is a finished product and can be run on some server and can be used by other people. The framework is not and it is hard to analyse the security of something that is not completely finished yet. It is yet unclear how the several

1Lines of code were counted using CLOC (http://cloc.sourceforge.net/) with a custom language definition to include unrecognised types of code used by Rails, such as ERB. The numbers indicate real code, and thus do not include blank lines and comments.
4.2. The approach

The approach we have developed uses a benchmark of test applications to analyse the security of web frameworks. In order to make this method general, we give a high level, abstract design of the benchmark instead of the actual implementations of these applications. This way the approach can be used for any arbitrary framework, since the design is framework independent. It states the requirements for the benchmark and presents some ways to achieve these requirements, but not the actual implementations of the solutions. The design of the benchmark is presented later in this chapter.

In order to be able to use the benchmark for an actual framework, say Ruby on Rails, the benchmark needs to be instantiated for this framework. This means that the high-level benchmark design needs to be translated to actual program code running in the web framework. The instantiation of the benchmark for a framework, is thus the same as implementing the benchmark design in that framework. It will result in an actual web application, or as we will see later multiple applications, built in that that framework. The implementation can be used to analyse the security of the framework. We describe the implementation of the benchmark design in Ruby on Rails in chapter 5.

The implementation of the benchmark by itself does not lead to a security analysis. To use the implementation to find security vulnerabilities, we use already available tools that scan the implementation for vulnerabilities. There are several options, but for reasons we will elaborate in chapter 6 we have chosen to use dynamic web vulnerability scanners [FO07, BBGM10, FVM07], which from now on we simply call dynamic scanners. These dynamic scanners can test web applications for many vulnerability types. They do this by performing requests and analysing the responses. They test the complete application stack, so also the framework the benchmark applications are built in. This means that a vulnerability found by the dynamic scanners in the implementation of the benchmark could indicate a vulnerability in the framework. However, the vulnerability could also be caused by the benchmark application, or by some other part of the application stack, such as the web server. To verify if an analysis result is caused by an actual vulnerability in the framework, manual analysis should be employed. If an analysis result is not caused by an actual framework vulnerability, then this is a false positive. The design of the benchmark should prevent these as much as possible. However, when analysing the results of the
scanners, the possibility of false positives should always be taken into account.

So to summarize, the developed methodology consists of the design of the benchmark and the described approach of instantiating the benchmark for a framework and analysing it with the dynamic scanners. If a security analyst wants to use the approach to analyse a new framework, then he should take the following steps:

1. Implement the benchmark design in the target framework. This leads to one or more web applications written in that framework.
2. Analyse the benchmark implementation using the dynamic scanners. This leads to analysis results, which could indicate a vulnerability in the target framework.
3. Study the analysis results and employ manual analysis to verify determine the actual vulnerabilities in the target framework. This leads to a list of vulnerabilities found in the framework, which could be used to give a verdict of the security of that framework.

A schematic overview of this approach is displayed in figure 4.1.

![Schematic overview of the steps an analyser should perform to analyse a new framework](image)

Figure 4.1: Schematic overview of the steps an analyser should perform to analyse a new framework

The rest of this chapter continues with the design of the benchmark, i.e. what are the requirements for it and how should it behave. The benchmark will have a modular design. We start by presenting the global design of the benchmark, stating the requirements and design choices applicable for every module. This is followed by the specific design for each of the implemented modules. We have chosen to implement two modules; one to analyse the framework for SQL injections and one for XSS vulnerabilities.

### 4.3 Global benchmark design

Web frameworks provide several functionalities to help developers protect their applications against common vulnerabilities. However, it is possible that a provided functionality is not completely secure and contains vulnerabilities. Furthermore, web frameworks often also provide many other, not security related, functionalities to ease developers building web applications. Although these functionalities have no specific security purpose, they could still introduce security vulnerabilities. We want to test all these functionalities and determine whether they have vulnerabilities or not. This way we can determine whether the web framework is “secure” or not.

We will now explain how this can be achieved by describing the design of this benchmark. This design should be as general as possible, such that it can be applied to any arbitrary framework. The design can be seen as a “guide” with instructions to create a benchmark for an arbitrary framework, such that we can test the security of that framework. For this reason the general design should not yet include any framework specific choices. These are saved for the implementation of the benchmark for a specific framework, and will be discussed later in this thesis. We will now state the goal of the benchmark. After that, we give the requirements this leads to, explain the design choices we have made and finally we present the global architecture of the benchmark.
4.3. Global benchmark design

4.3.1 Benchmark goal

The goal of the benchmark:

The goal of the benchmark is to provide an interface between the (security) functionalities provided by the framework and the dynamic scanners, such that the scanners can test those functionalities for security vulnerabilities.

The dynamic scanners cannot test the framework directly. They test the complete application stack (including the web server) and therefore need a working and running application. A framework cannot be run directly, but needs to be used by an application. Therefore, to use the dynamic scanners, we require some sort of interface on which we can run these scanners. The benchmark forms that interface. So basically the benchmark is an application (or multiple applications) written in the framework we want to test. We can run the scanners on the benchmark and this possibly results in the discovery of vulnerabilities. A vulnerability found in the benchmark application, should indicate a vulnerability in the framework.

4.3.2 Benchmark requirements

As stated above, the benchmark is an application in the framework. In order to make sure that it can be used to test the framework functionalities, it should meet certain requirements. We will now state and explain these requirements:

Benchmark requirement 1: include all provided security functionalities

To make sure the framework is completely secure, we should test all the security functionalities provided by the framework. What is not included in the benchmark cannot be tested by the scanners. Thus to find the most vulnerabilities in the framework, the benchmark should be as complete as possible. However, this might not always be feasible or possible, and it could conflict with other requirements, such as the requirement to limit false positives (requirement 4). Therefore, it is allowed to deviate from this, as long as every deviation is considered thoroughly. What in the end should be included in the benchmark and what not, depends much on the particular framework. However, we give some guidelines on what categories of functionalities to include later when discussing the modules specifically.

Benchmark requirement 2: use the functionalities we want to test

We want to enable the scanners to test some particular functionality through the benchmark. In order to do this, the benchmark should actually use that functionality. The benchmark forms the interface between the scanner and the framework functionality, so if the benchmark does not use a particular functionality, then the scanners cannot test it.

Benchmark requirement 3: use all the included functionalities in any way possible

Not only should the benchmark use the functionality we want to test, it should also be used in any way possible. Often, a functionality can be used in multiple ways or it can take several types of arguments. The benchmark should explore all these ways, so we can be sure the particular functionality is completely safe, no matter the way functionality is invoked or with what arguments it is called.

Benchmark requirement 4: limit false positive results during analysis

The requirements above all have to do with trying to increase the coverage of the benchmark in order to increase the amount of vulnerabilities found, or in other words, decrease false negative results. However, we also want to limit false positive results, since these distract the analyser from the real issues and it takes extra time to study them. Because of this, the benchmark should be designed in such a way that false positives would be limited whenever this is possible.

Benchmark requirement 5: ensure the functionality usage is accessible

The benchmark should not only use the functionality we want to test, but also make sure this usage can be invoked by the scanners. This means that it should be possible to externally trigger the execution of the code that uses the particular functionality. We will discuss later how this can be achieved.
Chapter 4. A benchmark approach

Benchmark requirement 6: provide guidance on the functionality’s access
The dynamic scanners should not only have the option to invoke the functionality, they also should have some guidance on how this can be done. They should “know” what functionality is included and how it can be invoked, otherwise the scanners will likely not test the functionality.

Benchmark requirement 7: make functionality parameters and options controllable
Not only the usage of the functionality should be triggerable externally, it should also be possible to control the parameters and options for that functionality. The scanners search for vulnerabilities by trying to perform attacks and analyse the response. To try these attacks, the scanners should be able to control the parameters and options for the functionality. It should be possible for the scanner to let an attack attempt end up in the functionality through the benchmark.

Benchmark requirement 8: provide guidance on the parameters and options
As it should be clear how to access the functionality, it should also be clear what the options and parameters for the functionality are and how these can be controlled. If this is absent, then the scanners have no idea of the several ways the functionality can be used and how the arguments can be set. They could of course try things at random (brute-force), but this is not efficient and it is likely possibilities will be missed. This reduces the coverage of the test. Therefore, there should be guidance on what data the scanners can send and how they should send it, otherwise the scanners will likely not perform the testing correct. Also some guidance on the expected type of the data is useful.

Benchmark requirement 9: output relevant responses of the functionality
As said before, the scanners test for vulnerabilities by performing attacks. In order to determine if there actually is a vulnerability, the scanners need to analyse the responses given after trying a certain attack. Therefore, the complete output of the functionality should be made available externally.

In some cases, the scanners also detect vulnerabilities by analysing messages of possible errors that occurred. Normally, in a live application, error reporting is disabled or limited. However, in our case, we are performing tests in an test environment we can control. Therefore, we should make it is as easy as possible for the dynamic scanners to detect vulnerabilities and enable full error reporting. By outputting both the result of functionality as well as possible error messages, the scanners will be best capable to detect vulnerabilities in the benchmark (possibly indicating a vulnerability in the framework).

Benchmark requirement 10: include a self sanity check
During our first tests we noticed the scanners did not find any problems at all. This was because of a bug in our benchmark application preventing error messages from being displayed. This indicates that it is possible that we miss actual vulnerabilities in the framework, simply because the benchmark is not correctly implemented. For this reason, we decided to add this extra requirement of a self sanity test that can be used to verify if the benchmark and testing setup are functioning correctly.

4.3.3 Benchmark design choices
We have established the requirements the benchmark should meet in order to ensure that the scanners can test the framework functionality. There are however still many ways to implement these requirements. To give some guidance on this, we have made several design choices that will be used in the global architecture of the design. We will now present and explain these choices.

Benchmark design choice 1: the benchmark will have a modular design
The benchmark will have a modular design. This way it is easy to extend the benchmark with new tests. Furthermore, we can enable and disable modules and test them separately. It is likely that the benchmark will be very large in the end and testing it completely with all the scanners can take a lot of time. In this case it is very useful if we can completely disable several modules to enhance testing times by only testing the module we are currently interested in. We have chosen to setup each main module as a separate web application within the framework. Each main module however can have several submodules which can be enabled separately as well. Again, this is useful for performance while testing, but for some modules it also has other benefits as we will see later for the SQLi module. The exact modules that will be used
4.3. Global benchmark design

depends on the categories of functionalities that will be included in the benchmark and will be discussed later in section 4.3.4.

Benchmark design choice 2: create a web page for each tested functionality

The benchmark is a web application created in the framework. Web applications usually consist of many web pages where a user can perform actions. The application contains code that renders the web page and handles the actions performed by the users. However, usually a web application also contains other code, for example for background scripts (cronjobs). As stated in requirement 2, the benchmark should use the functionality we want to test. To meet this requirement, we need to use the functionality provided by the framework somewhere in the code. We choose to use it in the code for the web pages, since we then also directly meet requirement 3 stating that the usage of the functionality should be accessible. Web pages are accessible to the users of the application, and thus also to the dynamic scanners.

So in order to make sure that we use the functionality and that this usage is accessible, we create a web page for each functionality. We make sure we include all the functionalities we want to test to meet requirement 1. This means we have a web page for all the functionalities. Requirement 5 states that we should also use the functionalities in all ways possible. To meet this requirement, we make sure that in the code we cover all ways a functionality can be used. This can lead to multiple web pages for a single functionality or otherwise, to some way to control how the functionality is used on the web page (more on this later when discussing design choices 6 and 7).

If desired, functionalities can also be grouped and made accessible on a single web page, as long as every functionality is used on at least one web page. The dynamic scanners can then perform requests on these pages and thereby invoke and possibly control the functionality. This way we make sure that all functionalities and all the ways they can be used are covered in the testing.

Benchmark design choice 3: also include “shortcut” functionalities

During the implementation of the Rails benchmark, we noticed that the framework provides many “shortcut” or aliased functionalities. These are methods that in fact call other (public) framework methods and preset some arguments and options. The benefit of these methods is that they help to keep the code cleaner, by providing a simple method for very common operations. It might be that the methods that these shortcuts call have already been included in the benchmark. Therefore it is not strictly necessary to include the shortcut methods separately, since the more general public method should apply the protection, unless clearly stated otherwise. However, we decide to include these shortcut methods anyway, for the following reasons:

1. It is possible that the framework later changes the internal workings of the shortcut method in such a way that they perform the actions themselves instead of delegating them to already tested methods. The main reason for this would be to improve performance. The shortcut methods are used for very common actions, and therefore there can be a huge performance gain when these methods are made more efficient. We want to be as safe as possible and make sure we still cover these methods when this happens and therefore include them separately.

2. It might not be directly clear which methods are shortcut methods and which methods perform the actions on their own. In order to determine this, you would likely have to dive into the internal workings and source code of the framework. This trouble can be saved by simply including all the public framework methods.

3. Including the shortcut methods is usually not much extra work. This is because the more general method that the shortcut method uses is already included. We can simply reuse the code for this test, make a few alterations and the shortcut method is included in the benchmark as well.

This decision is also more in line with requirement 1. Only if the framework has a lot of shortcut methods, which makes including them all infeasible, it could be an option to choose not to include them. However, this choice depends on the particular framework and is therefore left to the benchmark initiator.

Benchmark design choice 4: exclude “unsafe by design” functionalities and arguments

There could be functionalities provided by the framework that are related with a benchmark module, but are not secure for usage with untrusted input. For example, Rails offers a complete database interface,
which in many causes automatically protect against SQL injections. However, there are some methods that
do not apply this protection automatically, either because this is not possible, it is too difficult or not
efficient. These methods are deliberately unsafe and the developer using them should apply appropriate
protection himself. Logically, when including such functionalities in the benchmark, we would get lots of
positive results indicating a found vulnerability. However, we see these results as false positives, because
it is generally known that the particular functionality is unsafe. In accordance with requirement 4 we
therefore choose to not include these functionalities.

The same analogy is applied to partially safe functionalities. These are functionalities that are only safe
in particular circumstances or of which only some arguments are safe. We include these functionalities,
but only make the safe ways available to the scanners for testing. This way we ensure that the framework
is tested as much as possible, while limiting the amount of useless positive results.

To determine if a functionality is unsafe and thus should be excluded from the benchmark, we use
the framework’s documentation or other sources that state these things. When there is doubt, i.e. it is unclear
if a functionality is unsafe or not, we choose to include the functionality accordance with requirements 1
and 3. Increasing the benchmark’s coverage, thus preventing false negatives, takes precedence over
preventing false positives.

Benchmark design choice 5: create an index page with links to all web pages
According to requirement 4 the dynamic scanners should have guidance on what functionalities there are
and how they can be accessed. Dynamic scanners work by first mapping the entire web application by
indexing all the available pages. This is usually performed by spidering, i.e. the scanner starts at the
root of the application, finds all the internal links, follows these and then repeats the process for the new
found pages. So in order to make sure the scanner knows all the functionalities, the application should
have a link to all pages using the functionalities. We choice to create an index with all links to all pages,
and put it in the root of the application. This way we make it the scanner as easy as possible, and we
can be sure that all pages are found.

Benchmark design choice 6: use HTML POST forms for controlling the functionalities
arguments and options
Not only should it be possible to invoke the functionalities, it should also be possible to set and control
any arguments and options (requirement 5). Dynamic scanners have several options to do this, mostly
by setting the data or the headers for the HTTP request. They could for example change the user-agent
header or set the cookie-data header. We could use the data in these header fields to control the web
framework functionality, i.e. use the data from the header fields as the arguments for the functionality.
This way the dynamic scanners are able to control that functionality.

However, besides from being able to control the functionality, the scanners should also have some
guidance on how this can be done (requirement 5). If we use HTTP header fields for the controlling, then
the scanners are able to control the functionality, but they still have no clue on how to do this. They
do not know what data they should send or in what header fields they should send it. Therefore, we
choose to not use HTTP header fields for setting the data, but to use the data part of a HTTP POST
request. We can then use an HTML form (with method POST) to set this POST data and thus control
the functionality. This form also provides guidance to the dynamic scanners, because they can parse the
HTML page with the form and see what fields there are and even what type these fields have. This way
they know what data they can set and what the expected type of this data is. Another benefit of using
an HTML form for controlling the data, is that almost all dynamic scanners have support for testing web
applications by sending POST data or manipulating HTML forms, whereas this is not always the case
for other HTTP request fields, such as cookie-data.

Benchmark design choice 7: create a separate HTML form field with a sensible type for
each argument and option
We should make sure that every argument for the functionalities can be controlled and that the scanners
have the best guidance on the possibilities as possible. To do this, we choose to create a separate HTML
form field for each of the arguments and options. Furthermore, we should use a sensible type for this
field. If for example the parameter type is a string, then we should choose an HTML input field with
type text. However, if the argument should be an ID of an existing database record, then we should use
a drop-down field and fill it with the currently existing IDs, to provide the scanners with some guidance on possible values. This design choice corresponds with requirement 8.

**Benchmark design choice 8: respond with an HTML page with the result and errors of the invoked functionality**

Requirement 8 states that the complete output of the functionalities, including all the errors that occurred, should be made available to the scanners. As discussed earlier, we create a web page for each functionality. On this page, there is a HTML form that controls the particular functionality. After submission of this form, which results in an HTTP request, the functionality is invoked with the arguments and options from the form. The output and the errors this results in, should be send back as the HTTP response on the submission request. This is done in the form of an HTML page. The scanners receive this HTML page, can parse it and look for it for certain patterns or errors to determine if the tried attack on the functionality was successful or not.

**Benchmark design choice 9: deliberately use an unsafe functionality as self sanity check**

Following requirement 9 the benchmark should include a self sanity test to verify if the benchmark implementation and testing setup are sound. We implement this so called self sanity test by including a small submodule that employs some unsafe functionality. We know this functionality is unsafe, so if everything is working correctly, then the dynamic scanners should find a vulnerability (true positive) in this submodule.

### 4.3.4 Global benchmark architecture

We have established the requirements for the benchmark and made several design choices. We will now translate this to the global architecture of the benchmark. The benchmark is application, or multiple applications, written in the framework we want to test. These applications together should allow the dynamic scanners to test the security functionalities of the framework for vulnerabilities.

**Modular design**

The benchmark will have a modular design. Each of the modules will become a separate web application written in the framework. This way we keep a clear overview and it is easy to test the modules separately. The benchmark is the collection of all the web applications for the separate modules.

In order to be able to separate the modules this way, we need to ensure that the modules are composed in such a way that they are completely independent of each other. Otherwise, the modules cannot become separate applications and cannot be tested separately. Also, the functionalities tested in a certain module should somehow belong to each other, i.e. there should be reason why the functionalities are put in the same module. The categorisation for the modules will be discussed later in this section.

**Module architecture**

Combining all the requirements and design choices, we can summarize the design for each module as follows:

1. Each module in the benchmark becomes a separate web application created in the framework.
2. The module contains a particular set of security functionalities we want to test.
3. For each functionality we create a web page in the application with an HTML form.
4. The HTML Form contains a field for each option or argument the functionality supports.
5. The type for the field should be sensible, such that it provides guidance to the scanners.
6. The code that handles the submission of the form should invoke the tested functionality and translate the form’s POST data to the arguments and options.
7. The code should obtain the result and potential errors given by the functionality and display them as HTML page.
8. The module should have a special root page, which is an index with links to all other pages in the module.

Modules

What remains is to determine what functionalities we need to test and thus what modules to include in the benchmark. This will largely depend on the particular framework. However, we can determine what categories of functionalities that should be included in the benchmark. It appears that a particular framework functionality usually only protects or should be secure for a certain type of vulnerability. For example, if a framework offers the functionality of a complete database interface/ORM, then this functionality clearly should be protected against SQL injections. Therefore we categorize the functionalities based on the type of vulnerability it should (be) protect(ed) against. Since each functionality often is only relevant in respect of one type of vulnerability, they usually fall in only one category. Furthermore, the functionalities that fall in the same category, are independent from the functionalities in other categories and can be tested separately. Therefore, this categorisation can be used to compose the modules. Each type of vulnerability, becomes a functionality category, each category becomes a module and each module becomes a web application in the benchmark.

We determine the categories/modules to include by using the overview of common security protection mechanisms offered by web frameworks (section 2.2) and the overview of recent vulnerabilities in Rails (section 3.4) given earlier. Using these overviews, we can determine what types of vulnerabilities are important, present and relevant for Rails. Below we list the categories/modules we include in the benchmark, with a small motivation why these were chosen:

1. **SQL injection (SQLi) module**: many frameworks provide some form of protection against SQL injections, often included in a complete database interface/ORM. The benchmark should include an application to test this interface and determine whether it is really safe for use with input from untrusted sources. We choose to include SQL injections, because it is a very common vulnerability type in web applications, it can be very harmful and it is the second most common vulnerability in Ruby on Rails.

2. **XSS module**: many frameworks implement an automatic protection mechanism to prevent XSS vulnerabilities, usually by providing output escaping and offering safe helper methods to perform common output tasks. The benchmark should include an application to test this protection and verify if it indeed prevents XSS vulnerabilities. We choose to include XSS, because it also is a very common vulnerability type in web applications, it is relatively harmful and it has been one of the most occurring problems in Rails.

Due to time constraints we cannot design and implement modules for all the vulnerability types mentioned in section 2.2. Of these vulnerabilities, we believe SQL injections and XSS are the most interesting, mainly because of their large prevalence. To use the time available to us effectively, we have therefore chosen to focus solely on these two vulnerability types and corresponding benchmark modules.

Benchmark design overview

In the next sections we present the design and specific design choices for the SQLi and XSS module. However, before we continue we first present a schematic overview of the complete benchmark design in figure 4.2. This gives an idea of the structure of the benchmark and the modules and submodules it consists of. In the next sections it will become clear what these submodules are and why we have chosen to include them.

4.4 SQL injection module design

Following the global design, the basic idea for the SQL injection (or SQLi) benchmark application is simply to use all the provided public SQL methods, like query builders, SQL sanitization functions, et cetera. These methods should be used individually and in valid combinations with each other. The dynamic scanners should have access to these methods, so they can test them for SQL injections. The benchmark is most reliable if literally all methods are used and combined. However, this might be infeasible. If this is the case, then it will likely be enough to explore only the usual methods and combinations.
We distinguish two kinds of frameworks applicable to the SQL injection module:

A. The framework only supplies sanitization methods. An example of such “framework” is PHP. In this case we only expect to find vulnerabilities due to wrong sanitization code. These can be found by dynamic analysis tools.

B. The framework has a complete database interface, which builds the SQL queries and automatically applies SQL injection sanitization. The interface should provide methods to easily perform at least one of the standard database CRUD (Create, Read, Update and Destroy) operations, but preferably all of them. An example of an applicable database interface is Rails’ Active Record. For this category of frameworks we expect to find SQL injection vulnerabilities due to failure to apply the protection as well as vulnerabilities due to wrong sanitization code. The first can be found with both static and dynamic analysis tools and the latter only with dynamic analysis tools.

Both types of frameworks can be tested using the benchmark setup combined with dynamic scanners, but they require a somewhat different approach. However, we only discuss the approach needed for frameworks of category B. This is because these frameworks clearly are the most interesting to test for SQL injection vulnerabilities. Furthermore, in the frameworks of category A, there is too much freedom for the developer to decide how and when to use the protection mechanisms. Therefore, it is much harder to test these frameworks in a general and consistent manner. Frameworks of category B usually provide a complete interface for the database actions. This interface functions as an abstraction layer between the developers logic code and the database operations. Developers do not have to write SQL queries themselves any more. Instead, they call frameworks methods that will (eventually) build the queries for them, execute them and possibly return the results. In general these frameworks also automatically apply SQL injection sanitization. This reduces the effort for the developer and makes the prevention more robust, since it cannot be forgotten by the developer.

The basic idea to test these kind of frameworks is simply to use all the database methods provided by the framework and combine them in every possible way, as discussed earlier for the global benchmark design. However, there are some specific factors to consider when testing database interfaces and this leads to the following SQLi module specific design choices.

### 4.4.1 SQLi module design choices

**SQLi design choice 1: have a database system to use**

Since we are testing for SQL injections, we need to use a live database system during the testing. If the
framework supports multiple database systems, then you should test it with all of them. Sanitization code can depend on the database system and the interfacing module as well. However, testing with all databases might be too much work. If this is the case, it is also possible to test it only with the most common database system(s). Also the specific version of the database system could have an influence. In general, you should use the newest stable version that is supported by the framework. This is because we want to find security vulnerabilities in the framework, not in the database system and the newest version likely has the least security vulnerabilities itself.

**SQLi design choice 2: include base sanitization methods and CRUD functionalities**

The database interface supplies functionalities to interact with the database. Basically these functionalities perform a particular kind of query on the database. We can categorise these functionalities on the kind of action they perform on the database. We make a distinction between standard CRUD database queries and non-CRUD queries. Examples of the first category are simple “INSERT INTO ...”, “SELECT ... FROM ...” queries. Examples of the latter category are database management queries (“CREATE DATABASE ...”) and user management queries (“CREATE USER ...”). In live web applications queries of the first category are by far the most common. In fact, queries of the second category are so rare, that we choose not to include them in the benchmark.

Except for the database interface functionalities for the standard CRUD actions, frameworks often have the possibility to execute custom SQL queries as well. Usually the framework supplies sanitization helper methods to secure these queries too. We call these methods the base sanitization methods and they should be included in the benchmark as well.

In some frameworks it is possible that functionalities can be combined, for example to build a query piece by piece

If this is the case, then these functionalities should not only be included separately, but also in (valid) combinations with each other. Whether this is the case and what the valid combinations are that should be tested, is something the benchmark initiator should determine, since it is framework specific. It could be that testing all possible combinations is infeasible. In that case it could be an option to only test the most probable combinations.

**SQLi design choice 3: submodules for base sanitization methods and each CRUD action**

As discussed above we only include standard CRUD functionalities in the benchmark. However, each of the four CRUD actions are very particular and require a somewhat different approach to test. Therefore, we choose to distinguish between each CRUD action, and treat them as separate submodules in the benchmark. This has the additional advantage that we can easily enable/disable parts of the benchmark in order to focus the testing when running the complete module takes too much time. Another benefit is that some frameworks also make this (or a similar) distinction in their architecture to improve code cohesion. Therefore, there might be a connection between the modules of the benchmark and the modules of the framework, which makes it easier to track the responsible code once an SQL injection is found.

If the framework also includes base sanitization methods, as explained in the previous design choice, then these should be included as well in a separate submodule. Therefore, we have a total of five possible submodules, one for each CRUD action and one for the base sanitization.

**SQLi design choice 4: enable full error reporting**

The benchmark should enable full error reporting and all SQL errors that occur during the execution of queries must be displayed. This is actually already covered in global benchmark requirement but we want to stress its importance specifically here. This is because of the fact that dynamic scanners usually check for SQL injections, by checking responses for common database error strings. Often they also provide methods for detecting blind SQL injections (for example by differential analysis of the response, or by performing a timing attack), but these methods are somewhat less reliable and we want to make it the scanners as easy as possible.

**SQLi design choice 5: have a suitable database table to operate on**

The database interface functionalities should not only have a running database system to operate on, there should also be a database table in this system. The standard CRUD actions create, read, update or delete data in/from some table. This table should be created upfront before the tests can run and likely

---

2This is also the case in Ruby on Rails, where queries can be build by chaining so-called relation methods.
before the tests can be implemented, since the implementation depends on the columns in this database table. We should determine what this table should look like, i.e. what columns there are in this table. We want our testing setup to be as complete as possible. Therefore, the table should include a column for each supported data type. To determine the supported data types, we should determine what kind of framework we are dealing with. There are three options:

1. **The framework generates the database definition:**
   In this case the supported data types of the framework should be used. If the database system also supports other data types, then this will not matter, since (in normal use cases) these will not be used. If the database system does not support a data type supported by the framework and the framework does not use a fallback, then this data type should be skipped, since the application will not function otherwise.

2. **The framework generates code based on the database definition:**
   In this case the supported data types of the database system should be used. If the framework also supports other data types, then these should be skipped, since it is not possible to test them with this database system anyway. If the framework does not support a data type supported by the database system, then it could be interesting to still explore this type and see how the framework handles it. In this case it should either use a safe fallback or not function at all.

3. **The framework does not directly use the database definition:**
   This is usually the case for non-Active Record frameworks. There is no direct coupling between the database and the code. In this case you should test all the supported data types of the framework with all the relevant data types of the database system. For example, if the framework has a float data type, and the database system has both a float and a decimal data type, then both should be tested. The initiator of the benchmark should carefully decide which combinations of data types are relevant.

We do not only create the database table to operate on, but also prefill it with some records containing random data. We do this because some of the functionalities need data to operate on. For example in order to update a record in the database, there must be a record in the database that we can update. For other functionalities (for example functionalities that perform a general DELETE query) it might not matter if there actually is any data in the table. However, even for these functionalities it is best to test them with data, since this is more realistic.

**SQLi design choice 6: handle special arguments correctly**

According to requirement 8 there should be guidance on the usage of the functionality’s arguments. For the SQL injection benchmark, most of the arguments will be represented as simple text fields. However, there are two types of arguments that are special and require a special representation in order to guide the dynamic scanners. These two arguments are:

1. **ID arguments**: some methods take an ID or list of IDs as argument, an example is a read method that takes an ID and returns the corresponding database table record. In order to provide the scanners some guidance on the expected data type it is best to represent this field as a select or multiple select field, prefilled with the IDs of the records currently in the database.

2. **Condition arguments**: some methods take conditions (usually represented as the WHERE part of the query) in order to limit the data on which they operate. Conditions are applicable for read, update and delete queries. The best way to test the conditions argument is to include a field for each possible condition. However, a condition can be complex, since it consists of a field, an operator and a test value. This gives many possibilities. Furthermore, these conditions can be chained together with other (logical) operators. Testing all this, might be too much work. Because of this, we are satisfied with testing one condition for each column in the test database table. We choose for the equality condition, since it is applicable on (nearly) every data type. If more conditions are given at once, then these should be chained together using the AND operator. This way the affected database rows will rather be limited, which turns out to be particularly useful later delete actions (since otherwise to many or even all rows could be deleted at once).
4.4.2 SQLi submodules design

We will now describe the design of each of the five submodules. We have a submodule for the base sanitization methods and one for each of the four CRUD actions. For each submodule we will first describe the methods generally supplied by frameworks. After that we will discuss how these methods should be tested.

In this discussion we distinguish between object-oriented frameworks with an ORM database interface and non-ORM frameworks. This is a very broad distinction and there are still many differences within these categories, but it is too much work to discuss all in detail. We focus on the ORM frameworks, with a specific focus on the Active Record pattern, since that is used by our target framework, Ruby on Rails. Other ORM and non-ORM frameworks, will be discussed only broadly and there can be differences for specific database interfaces. Therefore specific attention should be payed when instantiating the benchmark for these kinds of frameworks.

Base sanitization methods

Web frameworks often supply a base sanitization method to escape values for use in standard SQL queries, for example in the conditions in the WHERE part of the query. The framework could distinguish between several parts of the query and provide different sanitization methods for each of them. This is for example useful for the ORDER BY part of the query, for which the value is a column name which is usually encapsulated with backticks (`) instead of single quotes ('). These situations require different sanitization functions. However, many frameworks (including Rails) do not offer these, because they believe those values should be checked against a whitelist, instead of sanitizing them.

The benchmark should include every base sanitization method and test it. Usually they only take one argument, the value to sanitize, and return it. The benchmark should make this argument available to the scanners. It should also make sure that the sanitized value is used in the query in the proper way, depending for which part the method is meant. Some of these methods can be used in multiple kinds of queries. It is likely not feasible to test every kind. Therefore, we choose to test them only in the most commonly used type of query. This usually is in the conditions part of a standard read query.

Create

The first CRUD operation is Create, which is responsible for the creation of data. It corresponds to database queries usually described with “INSERT INTO ...”. These queries insert a new row or multiple rows into a database table.

In Active Record pattern frameworks a create action is invoked by calling a specific method on a newly created object. Usually you create a new instance of a class, which is (for now) stored in volatile memory. Then you set the attributes on this object, in general by invoking setter methods. This data is stored in the volatile memory. Eventually, when the Active Record save method is called, an INSERT query is generated, which inserts the object in the database. The attributes are translated to database columns, an insert query is generated and executed. Frameworks using another ORM, usually insert data in a similar manner. Only the last step could differ. It could be that the saving is invoked by calling some method on the object, but it could also be that the saving is handled by a different object/class, as is the case for the Data Mapper pattern (see section 2.1.3).

In non-ORM frameworks data is inserted by calling some function or method stating the name of the table, optionally the columns to use and the values. The database name is usually provided as string. The columns can be provided in several ways, but usually as an array or hash of strings. The columns can be optional, in which case the function assumes you want to insert data for all the columns of the table. In this case you need to provide values for all the columns as well. A value could be blank, in the database default for that column or simply a blank value will be used. The values can be provided in several ways, most commonly as an array, hash or object. Some frameworks also support to insert multiple arrays at once, in which case an array of values will be provided as argument.

Of course, for all framework types all methods supplied to insert new records into the database should be tested, including all available options and arguments. In case of the create action, this usually is not very much, since INSERT queries do not support many options. Even if they do, these are often not supported by the framework, since in most common cases they do not provide a significant benefit.
In most cases the framework will only supply methods to insert a row in the database, which take the columns and/or data to insert as argument. Following the global design, this means we should create a form with a (text) field for each table column and thus each data type.

If the framework also supports to insert multiple \( n \) rows at once into the database, then this should be tested as well. At the very least this method should be tested for \( n \) is zero, one, two and many (for example five). For this we should create a form with separate fields for each of these rows. This means we get \( n \) fields for each of the supported data types.

Read

Read is the second and probably most used and most complex CRUD operation. It is responsible for the retrieval of data records from database tables or to perform calculations these tables (such as count and sum). The corresponding database queries are usually described with “SELECT ... FROM” queries. These queries retrieve one or more rows from the database table that match certain conditions and other options, such as the limit. In some cases data is selected from multiple tables at once, by using joins.

Methods for performing read actions in Active Record frameworks are usually called finder methods. They are almost always abstract class methods, since you do not have a object to perform an action on yet. The class usually determines the table to select the data from. The methods are very general, in the sense that they can be used to build any arbitrary select query. Usually these methods generate and execute the select query and support a lot of finder options. The finder options set specific parts of the query, such as the conditions for the WHERE clause or the ordering in the ORDER clause. Most of the frameworks automatically apply sanitization to the arguments set with the finder options. However, not all the finder options are automatically safe for SQL injections. For example, there could be an option to set the SELECT part of the query (i.e. the columns to select from the table). In many cases this option accepts a raw SQL fragment and it therefore is impossible to automatically apply sanitization.

In principle only a single finder method would suffice, if everything is controlled through the finder options. However, in practice, we see that there usually are several finder methods, where each performs a particular type of query, basically by presetting some finder options. These are examples of the shortcut methods as introduced in general design choice. Often, there are also separate methods to perform calculations on tables, since these queries return a single value, instead of objects. Furthermore, there often also are several methods to perform very common operations, with as prime example a method to quickly retrieve an object based on its ID. Again, other ORM frameworks have a similar way to perform read actions. There might however be small difference on the class/object responsible to retrieve the data.

In non-ORM frameworks read queries are actually performed in a similar manner. You usually have a method to perform a query, which takes many options/arguments. However, in this case, also the database table name should be supplied. There are also frameworks that do not supply any easy to use methods for read queries. In this case the developer has to generate the queries himself, using helper methods provided by the frameworks. All the supplied sanitization methods should then be used in at least the WHERE clause of a read query.

The benchmark should test all the supplied methods for the general read actions and include all possible finder options. Of course, functionalities and finder methods explicitly stated not safe for SQL injections could be skipped (SQLi design choice). For all methods that are safe, there should be a form, that contains a field, or in some cases multiple fields, to control each finder option. Read queries can take conditions in either the WHERE clause, the HAVING clause or both and these should be treated specially as discussed in SQLi design choice. Except for the general finder methods, also the methods for the very common actions should be tested. An example of this is a method to quickly obtain a record based on its ID. Note, for this method the ID field be treated specially as well.

3Actually, since Ruby on Rails version 3 the finder methods that have finder options as argument are being phased out. In the new setup, you create a relation object and call a (separate) method for each finder option, so called query methods. In the end an execute method can be called which will generate the query, taking the earlier set finder options into account, and return the results. More on this later in chapter when we discuss the Ruby on Rails implementation.
Update

The third CRUD operation is update, which is responsible for the updating or alteration of data records. It corresponds to database queries commonly described with “UPDATE ... SET” queries. These queries update a row of a database table by replacing the values for the columns with new values. In some cases multiple rows are updated at once.

In Active Record pattern frameworks, an update action is invoked by loading an object from the database (using one of the read methods), changing the attributes (using the setter methods) and calling the save methods. This will generate an update query that updates the current object, so one database row is updated at once. Except for the save method, also other object methods could update the object. However, in general these methods will also use the save method in the background. Furthermore, Active Record frameworks could provide other methods to update database rows. The main reason for this is that the object methods can only update one object at a time and this object needs to be loaded and initiated first. This is not very efficient when you want to update many rows at once. Therefore these frameworks often also supply other methods to generate a more general update query, which could have WHERE conditions and a limit. Obviously, both types of methods should be tested by the benchmark. Again, other ORM frameworks perform the update actions similar to Active Record frameworks and the only difference might be that some other object/class is responsible for the actual saving.

Non-ORM frameworks usually supply methods to update arbitrary rows in the database directly. These methods are best comparable with the latter type of methods discussed for the Active Record frameworks. They provide methods to update arbitrary columns of arbitrary database rows. The frameworks could also provide a (shortcut) method to update a particular row or particular rows of the database by specifying its/their ID(s), since this is very common. However, it is likely that in the background this method will call the more general update method and set a WHERE condition on the ID(s).

As for the other CRUD actions, the benchmark application should test all the supplied update methods. At the very least, there should be a field to update each of the data types (the columns in the test database table). If the framework contains methods to generate more general update queries, then this should be tested as well. These tests should include fields to set the options for these methods, such as the IDs, conditions or limit. These fields should be treated specially (SQLi design choice 6).

Delete

The fourth and final CRUD operation is delete. It is responsible for the removal of data records. Delete operations respond to database queries often described with the “DELETE FROM ...” syntax. This kind of queries will completely remove a row from the database, so that it cannot be found (read) again. It is also possible that multiple rows are removed in one query.

The most common way to perform a delete action in an Active Record framework is by loading the object from the database with one of the read methods and calling a specific delete or destroy method. This will create a delete from query, that deletes the object and corresponding table row based on its ID. However, as was the case for the update methods, for efficiency reasons frameworks often also provide other methods to delete rows from the database. These methods come in various forms. Most obvious (and probably most used) are the ones that take an ID or an array of IDs and remove the corresponding records directly from the database (without first instantiating an object). However, it is also possible that the framework provides a method to build a more general delete query, for example a method that takes conditions and builds a query that deletes all records that match these conditions (by setting the WHERE part of the query). Other ORM frameworks have similar delete methods, only the object/class responsible for the actual delete can be different, for example a specific database interfacing object is used to perform the delete.

In non-ORM frameworks the delete methods are usually of the latter sort that performs more general deletes. These methods provide a way to generate and execute general delete queries, that delete arbitrary database rows. They take at the very least an argument representing the (WHERE) conditions. Clearly this condition can be a simple equality condition on the ID. This is somewhat similar to the Active Record frameworks object delete method, although it is more efficient. This will likely be the most common use of the delete method. For this reason it is also possible that the framework provides a shortcut method for this use. Except for the conditions argument, it is also possible that the delete query is further bound by other arguments, such as a sort or limit option.
In principle all the supplied methods should be tested. However, the object delete methods in Active Record frameworks usually take no arguments. Therefore, testing if they are save for user input seems pointless. Of course the objects should first be loaded before they can be deleted, but this falls within the scope of the read tests.

All the other methods, both for Active Record frameworks as well as other frameworks, should be tested and there should be a form for each such method. This form needs to contain a field for each of the possible arguments and options. So for example, for the delete method that takes an array of IDs to delete, there should be a form with a field to select multiple IDs.

For the more general delete methods, the form should contain a field for each of the conditions and other options. This should be done in a similar way as for the more general update methods.

The need for data

There is one potential problem particular for delete actions, namely that at some point all data is deleted and thus there is no data left to test on. This is something we want to prevent, as stated in SQLi design choice 5. This problem is especially likely when the framework includes a general delete method that takes conditions as argument. When no conditions are supplied, the WHERE clause of the resulting query will be empty and thus all records in the table will be deleted. There are several ways to make sure the testing by the dynamic scanners is not hindered due to lack of records in the table:

1. Test the delete methods and especially the general conditioned delete method separately from the other CRUD actions. This way the testing of the other CRUD actions and the more restricted delete actions will not be hindered due to the fact that there is no test data any more. However, the testing of the general delete method can be hindered, since after performing one delete without conditions, there is no data to test the same method with conditions on any more. Although we believe this does not matter for finding SQL injections for this particular method.

2. Insert new data in the table after every delete action. Obviously, this way there will always be data to test the other methods on. One downside might be that the operations take a bit more time. Another potential downside is that the dynamic scanners could falsely believe that their actions fail, since there is not data deleted. However, most likely the scanners do not include such advanced features.

3. Insert data before every action. This is the opposite approach of the previous method. This solution has a larger performance downside, since for more actions an extra insert query is executed. However, it will not have the downside of “confusing” the dynamic scanners.

We have chosen to use a combination of the first and second method. For performance reasons and to enable more isolated testing we test benchmark submodules separately. While testing the delete module however, it is still possible we end up with no data after testing some functionality. This could influence the testing of other functionalities. Therefore, to prevent this, we make sure there is always data in the table by resetting the table to their original values after every delete action. During our experiments it appeared this does not hinder the dynamic scanners.

4.4.3 SQLi module design summary

Combining all of the above, we get the following design for the SQL injection module of the benchmark targeting frameworks of category B:

1. The benchmark should contain a table with columns for all supported data types.

2. The benchmark should use the supplied database interaction methods for standard CRUD actions as well as potential base sanitization methods.

3. The base sanitization methods as well as each CRUD action become separate submodules.

4. The benchmark should include a page with an HTML form for each functionality.

---

4 Which has several benefits such as that it is easier to analyse testing results.
a) These forms should contain a field for each of the possible arguments or options supported by
the method, or there should be a good explanation why some option is not supported.
b) This field should be a text field, except for the conditions and ID fields which are treated
specially.
c) The result of the query and potential database errors should be made available as response.

5. All pages should be accessible and referenced so dynamic scanners can find them and explore the
framework function through them.

4.5 XSS module design

Similar to the SQLi module, the XSS module should also test all the XSS related functionalities provided
by the framework. Functionalities of this kind include sanitization helper methods, automatic sanitization
mechanisms and other methods that could either help protect against or might possibly cause an XSS
vulnerability. Web frameworks protect against XSS in many different ways. Some examples of differences
are:

- The protection could be automatic or manual.
- The framework could employ input sanitization, output sanitization or both.
- The framework could filter dangerous XSS constructs by using simple search techniques or by
  actually trying to parse the content

These differences could influence the XSS benchmark module design and implementation significantly.
Fortunately, the larger and more popular frameworks are increasingly employing automatic protection
techniques, usually using output escaping. Since frameworks employing such XSS protection are the
most popular, we use it as base during the design and implementation of the XSS benchmark. Any made
deviation for other frameworks is clearly stated and briefly explained.

Using this type of frameworks as base we will now present the specific design choices for the XSS
module, such that it can be used to analyse all XSS related framework functionalities for vulnerabilities.

4.5.1 XSS module design choices

XSS design choice 1: include the base sanitization methods
Almost all web frameworks provide a number of sanitization methods that developers can use manually
or the framework applies automatically. We call these methods the base sanitization methods. They
are usually present in all framework types, regardless of whether they apply them automatically and
whether they apply them during input or output. These methods usually take a string, make it safe
for embedding in HTML or some other context and return it. Examples are a method that replaces all
special HTML characters with safe HTML entities and a method that filters all HTML tags not included
in a whitelist (preferably) or explicitly mentioned in a blacklist. Clearly, the benchmark should include
these methods and make sure they are tested thoroughly, since they form the base of the complete XSS
protection mechanisms of the framework. If the framework applies automatic protection, the benchmark
should test this as well. This can be done by outputting a user controlled variable directly in the HTML
of the web application so the scanners can verify it is sanitized correctly.

XSS design choice 2: include the view helpers
Web frameworks applying automatic protection through output escaping often also supply another form
of methods that should be included in the benchmark, namely the view helpers. View helpers ease
developers by performing common view (HTML) tasks. An example of a view helper is to format a
piece of text nicely in an HTML web page by replacing all line breaks (\n) with HTML enter tags (<br />
) or paragraph tags (<p>). Since these view helpers often need to return formatted HTML text,
they need to disable the automatic general protection. Clearly, this introduces points of possible XSS
vulnerabilities. For this reason input to these helpers should be properly sanitized using one of the
base sanitization methods discussed earlier. This increases the chance on an XSS vulnerability in a web
application significantly, because it adds a lot of locations the developer should protect. Fortunately,
many frameworks do this automatically by sanitizing input to these helpers before using it. This way developers do not have to worry about this themselves. However, developers should be able to trust on this automatic protection, meaning the framework should protect every helper that could introduce a vulnerability in a concise and adequate manner. This is something we should test in the benchmark, because as it turns out these view helpers often cause unintended vulnerabilities, as was the case for Ruby on Rails in the past. In order to do this, the framework should include all view helpers and test them thoroughly. In principle all possible arguments for these helpers should be tested by the benchmark, except the ones that are indicated to be unsafe (as discussed in general design choice).

It is possible that it is not feasible to include all view helpers. This could be because the framework provides a lot of such helpers, they have many arguments or can be used in many ways. If this is the case, it is possible to test only a subset of these helpers. A good option would be to only include the methods that disable the general automatic protection. These methods are much more likely to cause XSS vulnerabilities, since the other methods simply rely on the general protection mechanisms which which are tested separately. To determine if a helper method disables the general protection, the framework’s documentation or source code could be checked. This approach could likely limit the amount of helpers to be checked significantly. There are however some possible drawbacks. It could be that an unsafe method is left out by accident, which makes the benchmark less complete, i.e. it could miss vulnerabilities. This is especially likely if the framework’s documentation is not accurate or up-to-date. Another drawback is that the benchmark implementation cannot be used for new versions of the framework directly, since it is possible the implementation of helpers changes causing them to disable the automatic protection. This means that for every new framework version, the implementation of all helpers should be checked to verify if they still can be left out.

Frameworks that do not supply automatic protection can supply view helpers, but clearly these do not apply any protection and thus do not have to be included in the benchmark. Developers should sanitize the supplied arguments themselves, using the base sanitization methods, which are already included because of XSS design choice. Frameworks that apply input escaping also do not apply automatic protection in the view helpers, since they escape all unsafe data directly and thus trust that the input to every helper is already safe to use. Therefore, it is not necessary to include view helpers for these kind of frameworks either.

**XSS design choice 3: only test for reflected XSS**

As explained earlier in section there are three types of XSS: reflected, persistent and DOM-based XSS. For the benchmark design and implementation it matters which type of XSS we want to test for. We choose to only test for reflected XSS. Testing for DOM-based XSS is usually pointless, since these kind of vulnerabilities reside in client-side (JavaScript) code and server-side web frameworks often do not contain such code. DOM-based XSS protection mechanisms are also extremely rare in web frameworks. Furthermore, as we will see later in section not many dynamic scanners support (reliable) detection of DOM-based XSS.

As for persistent XSS, we do not include it, because for frameworks employing output escaping, the protection for reflected and persistent XSS is exactly the same. In fact, the protection mechanism usually does not even know whether the input was persisted (from the database) or direct (from the HTTP request). Testing for persistent XSS seems therefore pointless. Also, again, dynamic scanners seem to have inadequate support for checking for persistent XSS.

**XSS design choice 4: test in multiple contexts (context-sensitive testing)**

As explained in the context (location) is very important for XSS vulnerabilities. A particular string may be completely safe to use in HTML context, while it is unsafe to use in JavaScript context (inside a script tag). For this reason, we should not test only in one context, but in all applicable contexts. If a web framework only supplies one general XSS base sanitization helper, then it is very likely this helper will not be safe for every context. By testing it in all provided contexts however, it becomes clear for which contexts it is unsafe and thus for which contexts the framework supplies inadequate protection. If

---

5For example the CVE-2012-3463 vulnerability: [https://groups.google.com/forum/#!topic/rubyonrails-security/](https://groups.google.com/forum/#!topic/rubyonrails-security/)

6If in the future dynamic scanners do support DOM-based XSS or the target framework does offer DOM-based XSS prevention functionalities, then it might be an option to reconsider this design choice.

7If another XSS protection approach is used, then it might be sensible to include persistent XSS nevertheless.
a framework supplies multiple, context-sensitive sanitization methods, then we test each method in the applicable context, i.e. the context the framework’s documentation states the method is meant for. If for a certain helper it is unclear for what context(s) it is meant, then we test it with all contexts, or in other words: no context specified means it should be safe for every context. It might be tempting to test every method with every context, but this would lead to many false positives, which we want to prevent according to general requirement 4.

As for the view helpers, we test the view helpers only in the context it is meant for. We do this to avoid making the benchmark unnecessary extensive and complex and to avoid false positives. We determine the applicable context by checking the documentation and see if the context is mentioned. If not, then we can determine it by looking at the return value. If the return value contains HTML tags, then it is very likely meant for use inside the body tag. If it is absolutely unclear for what context the view helper is meant for, then we should test it in all contexts, and probably notify the framework developers that their documentation is incomplete or unclear.

In the next section we will present and explain the contexts that the benchmark should support.

**XSS design choice 5: submodules for view helpers and base sanitization methods**

The functionalities included in the XSS module, as discussed above, become separate submodules. So we have a submodule for the base sanitization methods, which tests all base sanitization methods in all the applicable contexts. We also have a submodule for all the view helpers. This submodule can be split further into parts for categories of view helpers. For example, many frameworks supply view helpers to build HTML forms, which usually are a bit more complex (and have to be used in combination with each other) than the other view helpers. It would therefore be nice to separate the submodule into parts for the form helpers and the simple helpers. Both parts could even be separated further based on the location where the helper is implemented in the framework (the name of the class/module/file it is implemented in).

### 4.5.2 XSS submodules design

Above we presented the general design choices for the XSS module applying to all submodules. This section discusses the specific design for each submodule. We have two main submodules: the base sanitization methods and the view helpers. We will discuss them both. However, as explained in design choice 4, we need to employ context-sensitive testing to both. Therefore, we will first discuss what contexts there are and how we can test in these contexts, before continuing with the submodules design.

**Common XSS contexts**

The basic idea of an XSS attack is to inject client-side code in the page returned by the web application, usually JavaScript code. However, a web page does not consist of a single programming language, but of several (HTML, JavaScript, CSS, ...). This leads to several contexts in which an XSS attack can be performed. The first step for us is to determine the (most) relevant contexts there are regarding XSS. Luckily, OWASP provides many relevant useful information on XSS, including details on how to handle different contexts. The second step is to determine how we can test the XSS protection mechanisms in these contexts. We will now discuss each common context and explain how to test it.

**HTML body context**

The most basic way to inject XSS is by injecting HTML in the body of the HTML page returned as HTTP response. This is called the HTML body context. Obviously, it is very easy to inject JavaScript code as well, simply include a HTML script tag with your JavaScript code. To test methods in this context, we should simply place their output in the body of the response of the benchmark application.

**HTML attribute context**

The HTML attribute context applies when untrusted data is used inside an HTML attribute, for example like `<input type="text" value="UNTRUSTED_DATA">`. This could be vulnerable to an XSS attack if it is possible to break out of the value attribute, in this case by injecting an double quote character ("). This way an attacker could insert his own attribute to the input tag and possibly execute JavaScript

---

48

[https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet)
4.5. XSS module design

code, by inserting an event attribute, such as onclick. He could also try to close the attribute and tag and insert his own tags, for example a script tag with custom JavaScript code.

To test methods in this context we should place their output in an attribute of some tag. Which attribute does not really matter, as long as it is not one of the special attributes we discuss below. Which tag does not really matter either, as long as it is a valid HTML tag that supports the special HTML attributes, since these are often used to execute XSS attacks. We choose to use a simple div tag in combination with the class attribute, which contains (between quotes) the output of the method we want to test.

Besides the standard HTML attribute context discussed above, there also are (at least) three special cases:

1. Untrusted data is used inside the style attribute. This effectively changes the context from HTML to CSS and thus should be treated as such.

2. Untrusted data is used in event attributes, such as onclick. This changes the context to JavaScript.

3. Untrusted data is used in URL attributes, such as href or src. By default these attributes are in the HTML context. However, it is possible for an attacker to switch them to the JavaScript context while this might be unexpected. This happens when the value of these attributes starts with “javascript:”, which indicates the JavaScript protocol.

We explain below how to test the first two cases. For the last case, we test it by creating an a tag with an href attribute and using the output of the method in it.

CSS context

This context applies when untrusted data is used inside CSS markup, for example inside the style tag in the document’s head section. A few browsers (for example Internet Explorer) allow to evaluate JavaScript expressions inside CSS using the expression function. Attackers could use this to execute XSS attacks. Almost all browsers also allow to switch to the JavaScript context for URL attributes, such as attribute-url using the “javascript:” protocol indicator.

To test in the CSS context, the output should be used in either a style tag or in the style attribute of some tag. We test both ways, where for the latter we choose to use the p (paragraph) tag.

JavaScript context

The JavaScript context is relevant when untrusted data is used inside JavaScript code, for example inside a script tag. In many cases this is very dangerous and implies an XSS vulnerability. There are however cases when untrusted data in the JavaScript context is not an immediate threat, namely when it is used inside a string encapsulated by quotes. We call this the JavaScript string context. Using untrusted data in this context is is safe as long as it is not possible to break out of the string context. Frameworks can provide methods to sanitize input for this context.

To test these methods in this context, we should include their output in a string inside a script tag. To include it in a string we should wrap it in quotes, either single quotes (‘) or double quotes ("). We choose to do both. Furthermore, to simulate real world usage, the JavaScript should be valid and we should do something with this string. We perform two actions: assign the string to a JavaScript variable, and use it in a function call. For the latter we choose to use the console.log function, since it takes a single string as argument.

Base sanitization methods

The base sanitization methods form the base of the framework’s XSS protection mechanism. They are responsible for sanitizing a piece of text, such that it can be used safely in various XSS contexts. They usually only take one input, the text to be sanitized, transform it into something safe and return it. Testing these methods is fairly simple. Just include them in the benchmark, in such a way that the scanners control their input. Their output should of course be made available to the scanners. How to do this exactly, depends on the context we are testing the method in. As discussed in design choice 4 the base sanitization methods should be tested in all applicable contexts. These are either the contexts the base sanitization method is explicitly listed to be safe for, or all contexts if this is not specified. In
the previous section, we already explained how to test a method for a specific context. So the output of
the base sanitization method should be used in the correct way for every context it is applicable to.

Two of the most common base sanitization methods available in almost all frameworks are a method
that replaces all HTML entities in some text by their safe HTML safe variants and a method that removes
HTML tags and/or attributes from a text. The first method is usually safe for both the HTML body
and attribute context, except for the three special cases. The latter is safe for the body context, but only
occasionally for the attribute context, depending on how it is implemented.

View helpers

Many web frameworks provide view helpers to ease developers performing common view tasks involving
HTML. Frameworks that provide automatic output escaping to prevent XSS have the task to allow these
view helpers to work without breaking the automatic protection. We want to test if these view helpers
indeed do not break the protection. To do this, we should include the view helper in the benchmark, and
make sure that their input can be controlled by the scanners. How many and what types of inputs there
are depends a lot on the specific helper. It is possible that not all inputs should be made available to the
scanners. For example, it could be possible that some input is used to specify a piece of HTML content
that is used directly in the output. Clearly, this input is not safe and should not be made available to the
scanners. As for the output of the helper, again it needs to be made available to the scanners in the correct way (as described earlier) taking the context into account. The
helper should be tested in the contexts it is meant for, or if this is unclear, in all contexts.

What view helpers there are is even more framework dependent than for the base sanitization methods.
Some frameworks include almost no helpers, while others, including Rails, include a lot for many different
tasks. Some even include complete form builders, to easily build HTML forms and handle things like
showing errors and displaying entered values. It is hard to say anything general about view helpers and
how to test them. This is too framework dependent and is therefore left to the developer implementing
the benchmark.

4.5.3 XSS module design summary

Combining all of the above, we get the following design for the XSS module of the benchmark targeting
frameworks performing automatic output escaping:

1. The benchmark should use all supplied base sanitization methods and all view helpers, unless
contradicted by some design choice.
2. The base sanitization methods and the view helpers become separate submodules. The view helpers
could be further divided into another set of submodules if desired.
3. The benchmark should include a page with an HTML form for each method.
   a) These forms should contain a field for each of the possible arguments or options supported by
the method, or there should be a good explanation why some option is not supported.
   b) This field should be a text field, except if scanners need specific guidance for the method.
   c) After submitting the form, the output of the method is returned in the correct way depending
on the applicable context(s).
4. All pages should be accessible and referenced so dynamic scanners can find them and explore the
framework function through them.
Chapter 5

Implementing the benchmark in Rails

We have implemented the benchmark in the Ruby on Rails web framework. Figure 5.1 presents an overview of the benchmark implementation and indicates which parts were implemented and which were left out. Of the SQLi module only the base sanitization methods were left out. This is because these methods are already covered in the methods for the CRUD actions. Writing custom SQL queries in Rails is rare, so developers do not often use them directly. For the XSS module the form helpers and some of the simple helpers were left out. The omitted helpers could be interesting to implement as well, but unfortunately we were not able to do this due to time constraints. Appendix 4 gives a detailed overview of all included and (purposely) omitted functionalities.

Figure 5.1: Schematic overview of the benchmark implementation in Rails. Green indicates that the part is (almost) fully implemented, orange indicates the part is only partially (half) implemented and red indicates it was not implemented at all. Cls, Rel and Obj are abbreviations for the Active Record method types as discussed in section 5.2.2.

The rest of this chapter discusses the implementation of the benchmark in Rails. We start by describing the configuration that formed the basis for each module. After that we discuss the module specific implementations, mainly focussing on the challenges that arose while implementing them. We conclude this chapter by evaluating the implementation.
5.1 The base application

The base for each module is simply a blank Rails application, generated by the rails new command. We have changed several of the default configuration options, in order to let the dynamic scanners function or for performance reasons. These changes are listed in table 5.1. Rails comes with three built-in environments (development, test and production), of which each can have their own configuration options. We have chosen to use the production environment for the analysis with the dynamic scanners. Therefore, for all the configuration options not mentioned, we have used the defaults for the production environment. Finally, the base setup also consist of a controller called home with a single action called index, which is also the root of the application. This page will contain the list of all tests included in the benchmark module, as described in global design choice 5.

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationController protect_from forgery call</td>
<td>removed</td>
<td>Make sure Rails’ CSRF protection does not hinder the scanners</td>
</tr>
<tr>
<td>config.action_controller.permit_all_parameters</td>
<td>true</td>
<td>Make sure Rails’ Mass Assignment protection does not hinder the scanners</td>
</tr>
<tr>
<td>config.consider_all_requests_local</td>
<td>true</td>
<td>Enable full error reporting (Global requirement [9])</td>
</tr>
<tr>
<td>config.log_level</td>
<td>:fatal</td>
<td>Only log very serious problem, to prevent flooding the log file</td>
</tr>
</tbody>
</table>

Table 5.1: Configuration options for the Rails applications

Besides these global options, the general setup also consists of a catch-all route. This is to prevent the logging of routing errors, which could flood the log file during scanning, especially since routing errors usually contain a large back-trace. The catch all route disable the logging and instead only render a simple 404 error page, that the dynamic scanners will further ignore.

A single benchmark application contains all the tests for one module. However a module can be split up in several modules, which can be separately enabled or disabled if needed. The chapter continues by discussing the implementation of each module in specific.

5.2 SQLi module implementation

The SQLi module is a separate Rails 4 web application, initialized as described above. SQL injection protection is almost fully implemented in Rails’ Active Record component. This component offers a few base sanitization methods and query builder methods for all CRUD actions. Many of the query builder methods automatically apply sanitization on the input values, but not all. In principle each safe method is included in the benchmark application.

5.2.1 Database system

According to SQLi design choice [1] we need a database system (RDBMS) to work with. Rails supports multiple database systems, but we have chosen one, PostgreSQL, to use for the benchmark implementation. We chose for PostgreSQL, since it is commonly used in Rails applications. It is easy to change the database system if desired, since Rails’ Active Record methods abstract from it.

The benchmark implementation also needs a database table to work on (SQLi design choice [5]). In Rails the framework generates the database definition. Therefore we create an Active Record model, called AllTypesObject, with an attribute for each type the Rails supports and let the framework generate the corresponding database table. Rails also provides a feature to automatically handle associations between model[1]. To test this feature as well, we create a second model called AssociationObject, which has just enough attributes to let the different association types function.

The structure of both database tables is presented in tables 5.2 and 5.3 respectively. For testing, we initialize these tables with some data. We create a total of ten AllTypesObjects and six AssociationObjects. The first AllTypesObject does not get any values at all, the second gets only blank values (empty

http://guides.rubyonrails.org/association_basics.html
string, integer 0, ...) and the others get normal values. Objects 4 to 10 will also be associated with the AssociationObjects using the different association techniques Rails provides.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type name</th>
<th>Ruby type</th>
<th>PostgreSQL type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td>Fixnum</td>
<td>integer</td>
<td>Automatically added by Rails</td>
</tr>
<tr>
<td>binary_col</td>
<td>binary</td>
<td>String</td>
<td>bytea</td>
<td></td>
</tr>
<tr>
<td>boolean_col</td>
<td>boolean</td>
<td>TrueClass False-Class</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>date_col</td>
<td>date</td>
<td>Date</td>
<td>date</td>
<td></td>
</tr>
<tr>
<td>datetime_col</td>
<td>datetime</td>
<td>ActiveSupport:: TimeWithZone</td>
<td>timestamp without time zone</td>
<td></td>
</tr>
<tr>
<td>decimal_col</td>
<td>decimal</td>
<td>BigDecimal</td>
<td>numeric</td>
<td></td>
</tr>
<tr>
<td>float_col</td>
<td>float</td>
<td>Float</td>
<td>double precision</td>
<td></td>
</tr>
<tr>
<td>integer_col</td>
<td>integer</td>
<td>Fixnum</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>string_col</td>
<td>string</td>
<td>String</td>
<td>character varying</td>
<td></td>
</tr>
<tr>
<td>text_col</td>
<td>text</td>
<td>String</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>time_col</td>
<td>time</td>
<td>Time</td>
<td>time without time zone</td>
<td></td>
</tr>
<tr>
<td>timestamp_col</td>
<td>timestamp</td>
<td>ActiveSupport:: TimeWithZone</td>
<td>timestamp without time zone</td>
<td></td>
</tr>
<tr>
<td>belongs_to_id</td>
<td>integer</td>
<td>Fixnum</td>
<td>integer</td>
<td>Used to test belongs_to association with AssociationObject</td>
</tr>
<tr>
<td>created_at</td>
<td>timestamp</td>
<td>ActiveSupport:: TimeWithZone</td>
<td>timestamp without time zone</td>
<td>Automatically added by Rails</td>
</tr>
<tr>
<td>updated_at</td>
<td>timestamp</td>
<td>ActiveSupport:: TimeWithZone</td>
<td>timestamp without time zone</td>
<td>Automatically added by Rails</td>
</tr>
</tbody>
</table>

Table 5.2: Schema for the all_types_objects database table. The type name is the name of the type used when generating the Rails migration. The Ruby type is the resulting type in the Ruby language the value is converted to when Active Record loads the object. The PostgreSQL type is the type in the PostgreSQL database.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type name</th>
<th>Ruby type</th>
<th>PostgreSQL type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td>Fixnum</td>
<td>integer</td>
<td>Automatically added by Rails</td>
</tr>
<tr>
<td>has_one_id</td>
<td>integer</td>
<td>Fixnum</td>
<td>integer</td>
<td>Used for has_one relation with AllTypesObject</td>
</tr>
<tr>
<td>has_many_id</td>
<td>integer</td>
<td>Fixnum</td>
<td>integer</td>
<td>Used for has_many relation with AllTypesObject</td>
</tr>
<tr>
<td>created_at</td>
<td>timestamp</td>
<td>ActiveSupport:: TimeWithZone</td>
<td>timestamp without time zone</td>
<td>Automatically added by Rails</td>
</tr>
<tr>
<td>updated_at</td>
<td>timestamp</td>
<td>ActiveSupport:: TimeWithZone</td>
<td>timestamp without time zone</td>
<td>Automatically added by Rails</td>
</tr>
</tbody>
</table>

Table 5.3: Schema for the association_objects database table.

### 5.2.2 Types of methods

Rails provides three different types of methods to perform SQL queries. We used these types to further distinguish the SQLi submodules, as can be seen in figure 5.1. Each type is called in a different way and results in different queries. The types are:
• **Object methods**: called directly on an Active Record object (representing some model, filled with data from the database). These are used to save objects to the database (create a record or update it) or delete it. To use an object query, the object should first be retrieved from the database using a read method.

• **Class methods**: called on an Active Record class (model). These are used to perform some query, when there is not already an object present. They can for example be used to create a new object, set its data and save it to the database in one step.

• **Relation methods**: called on an Active Record relation. A relation is a special object used to build a query. There are two types of relation methods, query methods and execute methods. The query methods are used to set an option for a relation, which corresponds to a particular part of the SQL query. They return a new (updated) relation, so they can be chained. An execute method, generates the final query, executes it and returns the results. The result depends on the type of query. For example, a read query results in an object or an array of objects. Relation methods are mostly used to read objects from the database (constructing an `SELECT ... FROM` query). However, there are also relation methods for create, update and delete. For convenience, every relation method is also implemented as class method, to get a relation to start with. Execute methods become separate pages in the benchmark, while query methods become separate form fields that can be set for each execute method.

### 5.2.3 SQLi submodules implementation

#### Create module
The Rails benchmark includes all create related methods, which we split up in class methods and relation methods.

#### Read module
All read methods are relation methods, but we can distinguish several types depending on their action:

- **Object finder methods**: retrieve one or more objects from the database.
- **Value methods**: query the database for some value, for example the number of objects in a database table.
- **Calculation methods**: perform database calculation operations, such as `sum` or `average`.
- **By SQL methods**: take an SQL query and execute it to find objects or determine some value.

All these methods are implemented in the benchmark. Furthermore, this submodule also implements tests for the various ways to set conditions, as will be discussed by the implementation challenges later (section 5.2.4).

#### Update module
The update methods are also fully implemented in the benchmark. We split these into the relation methods (used to update several objects at once), the object single update methods (used to update a single attribute of a single object) and the object multiple updates methods (used to update multiple attributes of a single object).

#### Delete module
The delete methods are divided into the relation methods which can be used to delete multiple objects at once and the object remove methods which are used to delete a single object (retrieved by a read method). All delete methods are implemented.

#### Injection module
The injection module acts as the self sanity check to verify the benchmark and scanners are working correctly. We implemented two tests. The first uses the unsafe relation query method `order` to set a value for the ORDER part of the query. This value is not sanitized automatically. The second test uses the query method `where` which is safe when used correctly, but not when the values are directly put in the query using string interpolation.
5.2.4 SQLi module implementation challenges

This section lists the implementation challenges we ran into during the implementation of the SQLi module. It starts with the general challenges and concludes with the submodule specific challenges.

General implementation challenges

Not directly clear what to implement

We should only include methods that are relevant for the SQLi module, but this is not always directly clear from the Rails documentation. For example, it is not always clearly indicated what a specific method does: generate a query, execute a query, only influence the generation of a query, or has nothing to do with the database at all. This causes some overhead when implementing the benchmark, since some extra research into the Rails framework needs to be done.

An even larger problem is that we should not include functionalities that are unsafe by design in order to prevent false positives (following benchmark design choice 4). However, in many cases it is not directly clear whether a method is actually safe. According to the same design choice we should only skip functionalities when there is no doubt they are unsafe, for example because it is explicitly mentioned in the documentation. However, when following this requirement strictly, we would include too many functionalities causing unnecessary false positives and increasing benchmark implementation and test time. Therefore we decide to relax this requirement slightly and also check external sources and the Rails source code in order to determine if a functionality should be included. Unfortunately this causes some overhead when implementing the benchmark and it makes it more likely to wrongfully not include a method, which decreases the benchmark’s completeness.

Other errors

The dynamic scanners detect SQL injections primarily by checking the responses of the web application for error messages generated by the database. However, it appears that these checks are often too general, seeing every database error as a potential SQL injection. Clearly, other non SQL injection related database errors cause false positives in the scan results. To overcome this problem, we have implemented a handle_exception method that is used as global exception handler. This method will check the exception against a whitelist (implemented in the safe_rescue_exception? method) and if it does not indicate an SQL injection, it will be suppressed.

Another technique used by the dynamic scanners is blind differential analysis. This technique performs two queries that should lead to completely different responses when there is an SQL injection. Rails error messages can cause problems in this case, since if an error occurs the response will be completely different and the scanners see this as a possible injection. Again, we prevent these errors using the handle_exception method if possible. In some cases it is also possible to prevent the error in advance by implementing specific logic inside the test. If possible, we do this in order to keep the list of (global) safe rescue exceptions as small as possible.

In appendix B we provide an overview of all global safe rescue exceptions.

Partially safe methods

There are some methods in Active Record, that are partially directly safe to use and partially not. These methods usually take two or more arguments, of which some are automatically sanitized for you, while others are not. In Rails the database column names are never sanitized automatically. You have to sanitize them yourself, if they can contain untrusted user input. For this reason, almost any method that takes a column name is partially safe at best. Take for example the update_column method, which takes a column name and a value and updates the corresponding database column with that value. This method is partially safe, the value is automatically sanitized for you, while the column name is used directly. We want to include these methods in the benchmark, but this could lead to false positives due to the unsafe column name argument. To prevent these, we manually check the unsafe arguments against a whitelist.

Relation methods

Rails provides many relation methods that can be used to build a query, by chaining them together. We have chosen to create a separate test for each execute method, since they cannot be combined. However,\footnote{Such as \url{http://rails-sqli.org/}, which lists all unsafe Rails Active Record methods.}
in principle the execute methods can be combined with any possible combination of query methods and
the same query method can even be called multiple times. This means there is an infinite number of
combinations to combine these. Of course we cannot include all these possibilities in the benchmark. For
this reason we have chosen to only set each query method once and in a particular order, to limit the
number of possibilities drastically. We believe this does not influence the results, since the options set by
the query methods are more or less independent and Rails always uses a fixed order to set these. The
only exceptions are the query methods where and having since these take conditions, as we will explain
next.

Condition options
Two of the query methods, where and having can be used to set conditions for the query. The values for
these conditions should be automatically sanitized. Rails provides multiple ways to set these conditions,
which makes these methods harder to test. In fact, there are twenty different combinations (options)
how the conditions can be set. As discussed above, the query methods are tested in combination with
each execute method. If we would test all condition options as well, in combination with both where and
having and for every execute method, then we would get a lot of combinations. This would make the
analysis drastically slow. Therefore, we have chosen to test every execute method with only one condition
option. Since it is possible that the other condition options are insecure, we test these as well, but then
separately. We test them in combination with a single execute method (to, a) using only the where query
method.

Submodule specific implementation challenges

Create: multi create
Rails has a functionality to create multiple database objects at once, by calling the create method with
an array of hashes containing the values for each object. We want to include this functionality in the
testing. A problem that occurs is that the amount of items that could be inserted at once is basically
unlimited, and each item could get its own set of values. This leads to an infeasible amount of possibilities
the dynamic scanners have to explore. To prevent this, the amount is limited to a maximum of three.
Furthermore, all items will get the same values, which means we only need to insert one set of fields in
the form, instead of one set for each item. We do not believe this influences the results, since the queries
generated in the end are more or less similar.

Create: relation methods
Some create methods are also implemented as relation method. However, strangely enough these methods
only use one query method, namely create_with. All others are simply ignored. For performance reasons,
we chose to exclude these from the form as well.

Read: dynamic find by
Rails dynamically creates finder methods for each attribute of an Active Record model. For example, the
AllTypesObject gets a find_by_id method as well as a find_by_string_col method. These methods basically
translate to a simple SELECT query with a WHERE clause, for which the input value is automatically
sanitized. We have implemented these methods using a drop-down field to choose the attribute name
and a simple text field to set the value. However, the scanners will manipulate with the select box and
also try other, non-existing attribute names. This leads to a NoMethodError in Rails, which can confuse
the scanners. To prevent this, we first manually check if the attribute is allowed and if not, we redirect
back to the form.

Read: undefined functions for calculate execute methods
Rails also offers methods to perform calculation methods, using the database system’s built-in functions,
such as \textit{COUNT} and \textit{AVG}. Most of these functions operate on a column, but are not defined for every type. Calling these methods for an undefined column could result in a database error, which scanners could misinterpreted as SQL injection when using differential analysis. Therefore, we should determine which functions can be called on what columns and make sure the drop-downs to select the column only contain the supported columns. Furthermore, it should be checked that the supplied column is indeed valid on the server side, because the scanners could try other values, not included in the drop-downs, as well.

\textbf{Update: multi update}
Rails also possesses an multi update method that takes a list of IDs and the new values, and will update the corresponding IDs with these values. This method gives similar problems as the multi create functionality and we solved this also in a similar way. Only this time there is no amount to limit, instead the IDs that could be chosen need to be limited.

\textbf{Update: single attribute update methods}
There are a some methods offered by Rails to easily update a single attribute of a database record in a particular way. These methods cannot operate on every attribute. For example, there is an \textit{increment} method that adds some value to an attribute and saves it. This method can only operate on attributes of a type that implement the add operation (numeric attributes). If called on other methods, this leads to errors, which could confuse the scanners. To solve this problem, we have created lists of accepted attributes for each of these methods, that are used to build the drop-down to select the attribute.

\textbf{Delete: reset database}
In \textsection 4.4.2 a specific problem with the delete submodule is presented, namely that at some point there may be an absence of test data. This section also discusses three potential solutions. We have chosen to implement a combination of solution one and two. This means we will insert new data after every delete action. We do this by completely resetting the database and refil it with the initial data every time something is deleted. Furthermore, we also separate the testing of each submodule, so the other submodules will not be affected by this problem.

\textbf{Delete: disable limit}
As discussed in \textsection 5.2.2 some relation execute methods do not allow certain query methods. This is the case for the \texttt{delete} and \texttt{delete\_all} methods, which do not accept the \texttt{limit} query method. To prevent errors, we removed the corresponding fields from the form to set the query methods.

5.3 XSS module implementation

The XSS module is also a separate Rails 4 web application, again initialized as described in section 5.1. Rails’ XSS protection is almost completely implemented in the Action View component. It offers base sanitization methods for several contexts, as well as a large set of view helpers. Due to time constraints we chose to not implement every view helper, but just enough to give an an idea of the effectiveness of the benchmark to test these helpers.

5.3.1 XSS submodules implementation

\textbf{Base sanitization methods}
The benchmark implementation includes all Rails’ base sanitization methods, i.e. the methods listed in section 3.3. This submodule also includes the automatic protection (for which Rails uses the \texttt{html\_escape} method). Every base sanitization method is tested in the contexts it is applicable for. For example, the \texttt{html\_escape} method is only tested for the HTML body and attribute context. On the other hand, the \texttt{escape\_javascript} method is only tested for the JavaScript string context.

\textbf{View helpers}
The view helpers are only partially implemented. We decided to split Rails view helpers into simple helpers and form helpers, which are meant for form building. The latter are not implemented at all. Of the first category we have only implemented some. Rails has split these helpers into separate modules,
depending on the type of operation they perform. For example, Rails has a URL helper module, which contains helpers to build URLs. We have chosen to follow this distinction in the implementation of the benchmark as well. Usually a helper is only meant for one context, which is most often the HTML body context. However, exceptions should of course be tested in all their applicable contexts, similar to the base sanitization methods.

Injection

Similarly to the SQLi benchmark we also implemented an injection submodule that acts as the self sanity check to verify the benchmark and scanners are working correctly. We implemented this test by reflecting a form field value, without the automatic protection, which is disabled by calling `html_safe`.

5.3.2 XSS module implementation challenges

This section describes the implementation challenges we ran into during the implementation of the XSS module.

General implementation challenges

Not directly clear what to implement

As for the SQLi module, again it is not always directly clear what to implement, because the Rails documentation was lacking. It is relatively easy to determine the base sanitization methods and view helpers there are. However, it is not easy to determine which methods disable the general protection and which arguments of these methods are safe and which are not.

Options for helpers

Many view helpers not only have (required) normal arguments as input, but often also take a hash of options. These are optional, but since they could control the output, we should test them as well. In fact, it turns out these options often contain vulnerabilities. However, it is not always directly clear what options there are and what they do. Furthermore, there are also dynamic options, for which there is no predetermined structure. An example is the often used `html_options` option, which takes a hash that can be used to set any HTML attribute for the tag created by the helper.

Submodule specific implementation challenges

Base: allowed tags and attributes for sanitize method

The `sanitize` method filters unauthorized HTML tags and attributes from an input text. Rails allows some safe tags and attributes by default, but it is possible to overwrite this. We want to test the `sanitize` method in multiple situations, depending on which tags and attributes are allowed. Since testing every possible combination would be infeasible, we have chosen five situations that we believe provide to be sufficient. These situations are: default tags and attributes allowed, a custom set of tags and attributes chosen by us, default value for tags, but no attributes allowed, default value for attributes but no tags allowed and nothing allowed.

Simple helpers: helpers with blocks

Rails view helpers often have the possibility to call them with a block: a piece code to implement some part of the functionality of the method. For view helpers this often consist of some string that is used somehow. Since these blocks could be used in the return value (that is marked as HTML safe) or influence the XSS protection, we should include the possibility to call these helpers with blocks too. For all helpers we implemented the block return a simple string. Therefore, we can simply let the block return the value of some form field that can be influenced by the scanner to test it. To test the method with block, we can simply add a field to the form. However, there are also methods that disable some arguments when used in combination with a block. To test these methods we have created separate pages for the same method, one where it can be called with block and another one where it can be used without block.

Simple helpers: translation helpers

Rails also offers translations helpers for internationalization (I18n) support. These helpers take a translate key and optionally a list of arguments that are used in the translated string. The helper determines the
corresponding translated string and fills in the arguments at the right places. The arguments used in the string could contain user input and thus allow an XSS attack. Therefore, we include these methods in the benchmark. However, these methods have several modes of operation. First of all, the arguments are dynamic depending on number of interpolation keys in the translated string and thus it is also possible to use it without any arguments at all. Secondly, it is possible to indicate a translation string as HTML safe, such that it is possible to use HTML in it as well. Third, it could be that the key is missing, causing the helper to display a message that the key is missing or to render the default value. This causes different scenarios and it could be that the translation helper is safe in one scenario, while it is unsafe in others. Therefore, we should test all these scenarios. We have done this by creating three types of translation strings and made separate pages with a form for them. Also, we included several pages testing what happens when keys are missing, both with and without supplying a default value.

5.4 Evaluation of the implementation

Since it is interesting to know how difficult and how much work it is to implement the benchmark in a framework, we will now evaluate the implementation of the benchmark in Rails. The required effort for Rails does not necessarily have to reflect to other frameworks as well, but it could be an indication. We start by presenting some general evaluation observations, after which we will present the estimated amount of work for each of the modules.

5.4.1 General observations

We have successfully implemented the both benchmark modules in Rails, following the design presented in chapter 4. Of this design we basically implemented almost every part. For the SQLi module we also believe that we have implemented many of the relevant Rails 4 methods, such that they can be tested for SQL injections. However, we also left out several functionalities or only performed very basic testing as discussed in appendix A. The XSS module unfortunately is only implemented partially, due to time constraints. Both implementations follow every general benchmark requirement and design choice, as well as the module specific design choices.

Unfortunately however, the implementation was far from trivial. Each of the implementation challenges mentioned above raised the amount of work needed for the implementation, because for each challenge a solution had to be found and implemented. Furthermore, there simply are a lot of relevant methods that needed to be implemented, for both modules. Each method takes time to implement and this makes implementing the benchmark a labour-intensive job. Furthermore, implementing the benchmark cannot be done by simply following the design. Quite a bit of framework specific research has to be done, for example in order to determine what methods to include and what not. It should also be clear in what ways these methods can be used and how they work. Especially this research part of the implementation took quite some time. It is also easy to make mistakes here and miss methods that should have been implemented or implement them incorrect. This could make our benchmark incomplete and less reliable.

5.4.2 Amount of work per module

SQLi module

We estimate to have worked for about hundred man-hours on the SQLi benchmark, including both research as well as implementation and optimisation. The final implementation consists of five submodules and a total of over eighty tested methods, of which a significant amount can be used in different ways. It also includes twenty tests for each of the different combinations to set conditions (the condition options). The final result is a Ruby on Rails web application that consists of about two thousand lines of code, of which a bit more than half is Ruby and the rest mostly is ERB (Ruby HTML)\textsuperscript{5}. There is also code in other languages, but those amounts are negligible.

\textsuperscript{5}Lines of code are counted in the same way as was done for the Ruby on Rails framework itself in section 1.1.
XSS module

The XSS benchmark was only partially implemented. It consists of three submodules and a total of almost thirty tested methods. We worked for about forty man-hours on it. The end result is a Ruby on Rails web application consisting of somewhat over thousand lines of code, of which again somewhat over half is Ruby code, and almost all the rest is ERB/HTML template code.

5.4.3 Evaluation conclusion

To conclude this evaluation we can state that implementing the benchmark design requires quite a bit of work and is absolutely not trivial. It requires framework specific research, mostly regarding what to include and what not, as well as quite some programming work. This is unfortunate, because it makes it less easy to use our approach on a new framework. It is of course possible, but it could take quite some time to implement the benchmark for this framework. It also requires a great deal of attention to make sure everything is implemented correctly and relevant functionalities are not missed mistakenly. These are actually drawbacks of our approach, which we will further discuss in chapter 8.

5.5 Availability of the implementation

Both benchmark applications are actual Rails applications which are open-source and available through GitHub:

- SQLi benchmark implementation: https://github.com/kreintjes/rails-benchmark-sqli
- XSS benchmark implementation: https://github.com/kreintjes/rails-benchmark-xss

We also released online working demos of the benchmark applications. These demos run in development mode, meaning it is possible to change settings and a log of the last queries is shown:

- SQLi benchmark demo: http://sqli.rails-benchmark.krjb.nl
- XSS benchmark demo: http://xss.rails-benchmark.krjb.nl
Chapter 6

Options for the benchmark analysis

We need some way to analyse the benchmark applications, such that we can detect vulnerabilities in the framework. This chapter presents the options available for this analysis, why we have chosen for the dynamic scanners, how they work, which dynamic scanners there are, which ones we have chosen and why.

6.1 Analysis options

There are many ways to analyse the security of a web application, manually, automated or even using a combination. As indicated in section 4.2 we want to automate the process and we want to reuse the work of others. This means we need to use some type of automated tool that is already publicly available. This section presents the main categories of options there are, the options we tried before choosing the dynamic scanners and explains why these options turned out to be unsuitable.

6.1.1 Main categories of analysis options

There are two main categories of tools generally used to analyse web applications for vulnerabilities: dynamic web vulnerability scanners and static code analysis tools.

Static analysis tools

Static analysis tools [CM04, WCC +95] are designed to analyse the source code of an application in order to find security flaws. They do this by employing techniques like simple searches for unsafe constructs (grep), type checking or more advanced methods like data-flow analysis, taint tracking, symbolic execution, program verification, model checking, et cetera. Static analysis tools apply white-box testing, meaning they test the internal structures or workings of the application, sometimes without knowing the functionality this leads to. They analyse code and the application does not have to be running or even have to be finished. Originally, static analysis was not meant for web security, but nowadays there are many static analysis tools that are specifically intended for web applications. These tools can detect most common web vulnerabilities, including SQL injections and XSS. Some examples of static analysis tools are: RIPS[^1], Yasca[^2] and CodeSecure[^3].

Benefits of static analysis tools are that they usually are fast, have good code coverage and provide nicely formatted output for developers. The largest drawback of static analysis tools is that they usually are language and/or framework specific. They analyse the source code of an application, which is written in a particular language and the tool usually only understands that language. This means we cannot find one single tool we can use to analyse all arbitrary frameworks, but instead we need to pick a specific tool for each framework.

[^1]: http://sourceforge.net/projects/rips-scanner/
[^3]: http://www.armorize.com/codesecure/
Dynamic scanners

Dynamic scanners [FO07, BBGM10, FVM07] apply a different technique to test the security of a web application. They operate basically in the same way a normal attacker would, i.e. simply by trying attacks. They take a running web application as basis, study it, try different attacks and check whether they succeed. Dynamic scanners apply black-box testing, meaning they treat the web application as an abstract object, testing its functionality without knowing its internal workings. Because they perform black-box testing, they are not specific for one language or framework. They can scan any web application, as long as it is running somewhere. Dynamic scanners are specifically meant for web security analysis, and hence can detect many web related vulnerabilities, among which SQL injections, XSS, other injection attacks, CSRF, file inclusion and response splitting. We elaborate more on dynamic scanners, explaining the internal workings and discussing their advantages and limitations, later in section 6.2.

6.1.2 Unsuitable analysis options

Before deciding to use the dynamic scanners, we first considered some other analysis methods. These options all are specific for the Ruby on Rails framework or the Ruby language, mostly using static analysis techniques. We now present all the options we considered, and explain why these were not suitable.

Rubyx

At first we looked at scientific research regarding Ruby on Rails security. There are two groups that performed research into this topic. The first group, consisting of researchers from the University of Maryland, have developed some tools for both Ruby and Rails. The first two tools bring static typing to Ruby (DRuby [FAFH09]) and Rails (DRails [ACF09]). Building on this research, they later developed the tool Rubyx [CF10], which is a symbolic execution engine that can be used to analyse Ruby on Rails web applications for several security vulnerabilities. This tool would likely have been a great candidate to analyse the benchmark applications and thus the Rails framework with. However, unfortunately the paper was already quite old (three years), and the tools introduced in it only worked with older versions of Ruby and Rails. It might have been a possibility to upgrade the tools so they also work with newer Ruby and Rails versions, but unfortunately the authors did not make it available. They stated “DRails and Rubyx were quite fragile to begin with, and they’ve decayed to the point that they’re not usable any more.”, so probably upgrading them was not a feasible option anyway.

GuardRails

The second group, from the University of Virginia, also developed a Ruby on Rails security tool. This tool, called GuardRails [BMW+11], unfortunately suffered from the same problems. The tool itself supports Ruby 1.9, but some gems it requires only support Ruby 1.8. The paper indicates the tool is only Rails 2 compatible yet, but as far as we can tell there still is no Rails 3 version and also no active development. However, even more important is that GuardRails is more of a security policy enforcer for Rails web applications, rather than a tool that can be used to detect vulnerabilities, neither in an application nor in the framework. So this tool would likely be unusable to us anyway.

Brakeman

Seeing the other two tools were slightly outdated, we decided to look specifically for an up-to-date tool. This resulted in Brakeman[4], an open-source static analysis security scanner for Ruby on Rails. This tool is an community product, rather than the result of scientific research. It can detect a whole range of vulnerabilities including SQL injections and XSS attacks. Brakeman scans the code for common security problems, and reports these. We thought Brakeman would be a good candidate, but unfortunately the tool uses a rather simplistic scanning method. Even worse, it simply assumes the framework methods are secure. This means that it will detect problems due to insecure usage of framework methods, but not within these methods themselves. For example, Brakeman will complain about the code Model.where("column = '#params[:value]'"), because it directly uses a user supplied parameter (value) in the WHERE part of a query. If one would implement this using placeholders as follows

\[\text{http://brakemanscanner.org/}\]
6.1. Analysis options

If you call `Model.where("column = ?", params[:value])`, then Brakeman would consider it secure, since Rails should automatically apply the sanitation for you. Brakeman does not check whether Rails actually does this, it simply assumes this is the case. Clearly, Brakeman is not a good option for us to perform the analysis, since we want to find vulnerabilities in the framework, and not (only) in the benchmark application.

There are other community security tools for Ruby on Rails, but Brakeman seems the be the most popular and a quick study indicates these are not suitable either.

Analysis options for Ruby

After considering the Ruby on Rails specific options and concluding these would not work, we decided to try a different approach. We realized that every web application written in Rails, combined with the framework, is basically a Ruby program. There are tools to analyse the security of Ruby programs, just as there are for Java or C++. These tools could for example employ data-flow analysis or taint tracking to search for SQL injections and other vulnerabilities. We might be able to use these tools to test the benchmark applications for vulnerabilities. Since we are testing on the level of a Ruby program instead of a Rails web application, we are in fact testing the combination of web application and Rails framework. This means, that when we find a vulnerability, this could well indicate a vulnerability in the framework. However, unfortunately there also are some problems when trying to apply these tools for our purpose:

1. A Rails web application might in principle be a Ruby program, but it is no ordinary Ruby program. This means there are some issues to consider, such as that a Rails web application does not run stand alone, but is usually run by a web server.

2. A Rails web application has several sources of untrusted input. This is input supplied by the user of the web application, usually through GET or POST requests. In Rails, a special hash called `params` is automatically filled with this input. However, the Ruby scanners likely do not recognize this hash correctly. They likely do not know it contains untrusted input that may not end up in interpreters when it is not sanitized.

3. Ruby specific analysis options are likely not suitable for web specific security vulnerabilities. For example, ordinary Ruby programs could interact with a database as well, so likely these analysis options could check for SQL injection related problems. However, vulnerabilities like XSS or CSRF are web specific and likely not taken into account.

Because of these problems we decided to drop this idea and look into the dynamic scanners instead.

6.1.3 Motivation for dynamic scanners

We have considered four options, before continuing with the dynamic scanners. As indicated above, all these four options, had one or multiple problems, making them unsuitable for our needs. Dynamic scanners do not have these problems. Dynamic scanners automate the analysis and are widely available, fitting our basic requirement. Furthermore, they are specifically intended for web security and thus are able to check web applications for many common types of vulnerabilities. A problem with the first two Ruby on Rails specific options was, that they were outdated and therefore not suitable for newer Rails and Ruby versions. Since dynamic scanners are framework and language independent, they do not have this problem. The problem with Brakeman was that it makes the simple assumption that the framework is safe, and thus only checks the application. This problem does not exist for the dynamic scanners either. They perform black-box testing, and thus cannot make any assumptions about the framework. In fact, they cannot even distinguish where the application ends and the framework begins. Due to their nature, they test the complete application stack, including the framework. This means that they are capable of finding vulnerabilities in Rails as well.

However, the main advantage of the dynamic scanners is the fact that they form a general analysis solution. As indicated, the dynamic scanners can operate on any language and framework. This means that now only the second step of our approach, implementing the benchmark in the framework, is framework specific. The first and third step, respectively the design of the benchmark and the security analysis, are now framework independent, which even further generalizes our approach. This is a huge
benefit contributing to a more general approach for web framework security analysis. It might even allow us to compare security results of different frameworks, although this should always be done cautiously. Of course dynamic scanners also have some limitations and drawbacks, which we discuss in the following section.

6.2 Dynamic security scanners

Dynamic web application scanners can be defined as:

An automated tool, that performs black box penetration testing on a web application [TS10].

As elaborated above, we have chosen to use these scanners for our analysis, and this section will discuss them in more detail. It starts by explaining how they work and how they can be used, then it discusses some of the benefits and limitations of dynamic scanners and it concludes by presenting some of the scanners there are and how we choose the specific scanners to use further.

6.2.1 Internal workings

A dynamic web application scanner can be defined as an automated tool, that performs black box penetration testing on a web application [FO07]. Dynamic scanners employ interactive scanning, they actually try attacks against the web application. They operate basically like a normal attacker would: they study the application, they attempt attacks and they determine whether these worked or not. To perform these three tasks, a dynamic scanner usually consist of two main components, a crawler and an auditor, of which the latter performs two tasks. Besides that, they usually have several secondary components helping with other tasks. We discuss these components briefly.

Crawler

The crawler is responsible for the studying of the target application, and determines the possible attack vectors. First, it builds an index of all pages and files the web application consists of. It does so by starting at the root URL (which is entered manually), retrieves the page and analyses it for other URLs. If it finds URLs, it will repeat the process for these new locations, until the complete web application is discovered. Figure 6.1 gives a graphical overview of the crawling process.

![Figure 6.1: Overview of the crawling process performed by dynamic scanners [TS10].](image)

Second, it determines the possible inputs for an attack for each of the locations. Types of inputs usually supported by dynamic scanners include:
• **Forms**: HTML GET or POST forms. Each form field (also hidden and disabled fields) is an input. The crawler should analyze the HTML of the location and extract the form fields.

• **Links/URLs**: links with GET parameters in the query string. Each parameter forms an input. The crawler should analyze the URL(s) of the location and extract GET parameters.

• **Cookies**: cookies are used to store data on the client’s local computer. They can be altered and thus form an attack point as well. The auditor should check which cookies the application has set on the local computer. Unfortunately, the crawler cannot determine whether a certain cookie is used at the current location, so it needs to try all.

• **HTTP headers**: http headers contain request information, such as the user-agent of the client’s browser. These can be altered as well and might form an attack point for application that use them. The HTTP header attack inputs are usually determined by a pre-defined list. Again, the crawler cannot determine beforehand if a certain header is used at the current location, so it needs to try them all.

The combination of locations and inputs for that location are the attack vectors. These should be checked by the auditor. The crawler is an important first step. Without it, the structure of the web application is unclear and it is not known what pages there are that should be checked for vulnerabilities. Without the crawler, the auditor would not know what to do. Furthermore, if the crawling process is incomplete, then it is likely attack vectors are missed, increasing the chance of missing vulnerabilities.

**Auditor**

The auditor (or fuzzer) is responsible for finding the vulnerabilities in the pages discovered by the crawler. It does this by performing two tasks: trying an attack and checking whether it succeeds. For the first part, the auditor needs to perform an (HTTP) request to the application. For the second, it needs to analyse the (HTTP) response. After determining the possible attack vectors, the auditor can start trying attacks. Usually the auditor consists of several subcomponents, one for each type of vulnerability (SQLi, XSS, ...) it can detect. Each subcomponent can try several attacks to determine if the application is vulnerable. The auditor performs the following steps for each attack vector and attack it wants to try:

1. **Determine applicable inputs for location and attack:**
   From the earlier identified attack vectors, determine the inputs that are applicable for this kind of attack. In most cases all inputs are applicable, but for example for file upload attacks only file input fields are applicable.

2. **Generate attack input:**
   Generate the input data needed to try the attack. Usually the submodules have a static list of attack data they want to try. However, in some cases the attack data could also be generated dynamically or the auditor could be self-learning. The attack input data does not have to be a real attack that actually does damage, but could be something only used to identify whether there is a real vulnerability. For example, for SQL injections the auditor could try to input only a single quote (’), which can be sufficient to determine if there is a vulnerability.

3. **Execute the attack:**
   Execute the attack by performing an HTTP request to the target application containing the input data determined in the previous step.

4. **Analyse the response:**
   Retrieve the HTTP response and analyse it to determine if the attack succeeded. Continuing on the previous SQLi example, this would be checking whether the response contains a database error. This is because inserting a (extra) single quote in a query that contains an SQLi vulnerability, often leads to a database error.

5. **Determine exploitability:**
   After analysing the response the auditor usually knows whether the attack succeeded or not. It could then further investigate it, determine the type of attack, the probability that it is a real attack (and not a false positive) and more. After this is done, the auditor should log the result somewhere.
Chapter 6. Options for the benchmark analysis

Other components

Besides these two main components, dynamic scanners often have secondary components for supportive tasks. Some examples are:

- **Platform fingerprinter**: identifies the platform (OS, web server, programming language, framework) used by the target application. This can be helpful to perform more specific auditing, which could lead to more accurate results and save audit time.

- **Recon**: passive form of auditing which does not actually execute attacks, but only analyses locations, to check for potential security problems. An example is checking if there are hard-coded passwords in client-side JavaScript code.

- **Evasion**: enables evasion techniques to prevent the application from detecting or blocking the scanning. An example would be to include random delays between each request.

- **Auth**: module to authenticate at the web application using various techniques. This requires the user to supply valid credentials (i.e. it does not try to crack passwords).

- **Reports**: generates and outputs scan reports in various formats, such as HTML, XML, JSON, text or e-mail reports.

- **GUI**: originally dynamic scanners were command-line tools, requiring technical know-how to operate them. Nowadays, more and more scanners include a GUI to use them, making them available to a larger public. This GUI can be used to configure and start scans, as well as to analyse reports. Often the GUI is in the form of a desktop application, but also other forms, including web applications, exist.

6.2.2 Detecting SQLi and XSS

The basic working and structure of the scanners is now clear. We will now elaborate more on the techniques dynamic scanners can employ to detect SQLi and XSS vulnerabilities. We do this only for these two vulnerability types, since these are the only two implemented in the benchmark.

Detecting normal SQLi

The most basic way to detect SQL injection vulnerabilities employed by the scanners is to send strings that are likely to cause a database error if there indeed is an SQLi vulnerability. The scanner simply tries to send strings containing special SQL characters, such as single quotes (’), double quotes (“), backticks (`) and parentheses. Some examples of strings used by actual dynamic scanners are: "d'z"0", "'--" and "). These strings will cause errors if the web application uses them directly in an SQL query, without proper sanitization. The scanners study the HTTP responses and determine if they contain database error messages. They do this by comparing the responses against a list of database error messages of common database systems and frameworks. Clearly this detection technique only works if the application fully reports database errors and if the scanner recognizes the error format. These are two huge drawbacks of this detection mechanism, and therefore dynamic scanners often also employ techniques to detect SQL injections “blindly”.

Detecting blind SQLi

There are two techniques to detect blind SQLi injections. The first technique uses differential analysis. The scanner then sends combinations of inputs to the web application. Each combination consists of two values: SQL constructs of which one will evaluate to false, while the other evaluates to true. If there is an SQL injection, then these two constructs will likely cause the web application to produce very different results. This can be detected by the scanners using differential analysis in combination with an (user-determined) equality limit that determines how similar the pages have to be to be determined equal. If the results differ significantly, then there might be a blind SQL injection. An example of a combination of inputs used by a real scanner is: “1 OR 1=1” (always true) and “1 AND 1=2” (always false). If for example a web application uses this input in a query that retrieves a single object by its ID,
then the first will lead to one result, while the second leads to no results. This causes two very different responses, which is detected by the scanner and listed as a possible blind SQL injection.

The other technique to detect blind SQL injection timing attacks. With this technique the scanners send an input that causes the web application to take very long time to respond if there is an SQL injection. This is done by generating SQL construct that causes delays in the database system, mostly by using the `sleep` function. This causes the database system to sleep for some period before answering the web application. On its turn, this also causes a delay for the web application to respond to the scanners request. Following this rationale, it is very likely the web application contains an SQL injection if it takes very long to respond after receiving such input. There is one drawback of this approach, namely that the SQL constructs for sleep statements depend very much on the used database system. This means the scanner has to have a list of inputs to try for each database system it supports. This means that if the scanner does not find a SQL injection timing attack vulnerability, then it could very well mean the application simply uses a non-supported database system. An example of an input string used by a real scanner (for the PostgreSQL database) is “pg_sleep(TIME)--”.

Detecting standard XSS

The simple way scanners detect reflective XSS vulnerabilities is by trying to insert a custom HTML tag in the response body (the HTML body context). If this tag is present in the body and is not encoded, filtered or sanitized, then there is a XSS vulnerability. An example of an attack payload used by an actual scanner is `<some_dangerous_input RANDOM_DATA>`. There are two ways to detect if this tag is present in the response body: (i) the scanners could employ a simple search or (ii) the scanners could parse the body and see if the tag is present in the Document Object Model (DOM). The latter is more reliable, since it simulates the browser’s interpreting of the response.

Detecting XSS in other contexts

Besides the most obvious XSS vulnerability described above, there are also other possibilities in other contexts as we discussed earlier in section 4.5.2. Good scanners should recognize different contexts and perform special attacks for them. Also it should take the context into account when determining whether there is an actual vulnerability or not, otherwise this could lead to false positives. We will shortly describe detection techniques for the most common contexts:

**HTML attribute context**

Scanners can test for the attribute context by inserting strings that could possibly create a new attribute, such as "dangerous_new_attribute="RANDOM_DATA. To verify if there indeed is a vulnerability, the scanner would need to parse the response and inspect the DOM (a simple search might also be possible, but is much less reliable).

As mentioned earlier, there are also some special cases. The scanners should recognize these cases and test them as well. The `style` attribute and JavaScript event attributes, can be tested in the same ways as the the CSS and JavaScript context respectively, which we will discuss later. The URL attributes can be tested by checking if is it possible to inject the string `javascript:RANDOM_DATA` at the beginning of an URL attribute.

**CSS context**

Scanners can detect XSS vulnerabilities in the CSS context by first determining whether it is possible at all to inject some data inside the CSS context. If this is the case, then the scanner should test whether it is possible to switch to the JavaScript context. For example by using the `expression` function (IE only) or the `javascript:` protocol in URL attributes.

**JavaScript context**

To detect XSS in the JavaScript context scanners could try to insert some random data and see if it pops up inside a script tag or an other JavaScript context location. The best way to do this, is by parsing the response body. However in some cases it could be perfectly safe to use data inside JavaScript, for example if it is used in the string context (encapsulated by quotes). Scanners should take this into account to limit the amount of false positives. They could do this by checking if their injected data is encapsulated by quotes, and if so they should try to break out of these quotes. If this is possible, then there is an XSS
vulnerability. If not, then there is no vulnerability. To enhance JavaScript context XSS vulnerabilities detection, scanners could include a JavaScript runtime engine to actually evaluate the result and see if the attack succeeds. This leads to much more reliable results, but decreases scan speed.

Detecting non-reflected XSS

Up to now we only talked about detecting reflected XSS. There are however also two other types of XSS: persistent XSS and DOM-based XSS. With persistent XSS, the attack is stored and displayed by the application on some other page. To detect persistent XSS, the scanners should split the inject and check stage. They should first try to perform XSS attacks, and then revisit all pages to see if the attack payload can be found. However this method is not perfect and therefore not many scanners support it.

The other type, DOM-based XSS, is also hard to detect. One technique uses a simple search that scans the bodies of the web pages. This technique searches for dangerous JavaScript functions call that could allow DOM-based XSS, such as `document.write`. If these function calls are found, then the scanner checks if the arguments list contains user controllable DOM variables, such as `document.URL`. If a match is found then it means that an unsafe variable is used directly in an unsafe function, causing a DOM-based XSS vulnerability. In principle this approach works, but it is by far incapable of detecting all vulnerabilities. If for example a user controllable DOM variable is used indirectly as argument to a dangerous JavaScript function, then the grep search approach will not detect this. This could for example happen when the developer first assigns the DOM variable to a normal JavaScript variable, possibly applies some operations to it and then uses it in a function call.

In order to detect DOM-based XSS more reliable, the scanner needs to have a JavaScript runtime engine and actually execute the JavaScript. This way the scanner can trace the unsafe DOM variables through the program and see if they end up in an unsafe JavaScript function (taint analysis). If this is the case, then there is a DOM-based XSS attack. This approach is much more reliable, but unfortunately hard to implement and slow. Since both techniques have significant drawbacks, not many scanners support DOM-based XSS.

6.2.3 Usage and configuration

The usage of dynamic scanners is quite simple. For the basic operation, all you have to do is to configure the start URL of the application. After the scanner is finished, it reports all the potential vulnerabilities, which the user should manually verify. Although the basic operation is quite simple, the scanners often have many configuration options that should be carefully set to achieve the best results. Often (sub) components can be enabled and disabled separately. This way it is possible to for example scan only for SQL injections. There are also some global configuration options, such as the maximum amount of concurrent HTTP requests that may be performed. Besides this, there often also are component specific configuration options, which we will briefly discuss.

1. **Crawler**: has options to limit the depth, to limit the amount of links to follow and whether it should follow links to subdomains. These can help to determine the extensiveness of the crawling and prevent it from running indefinitely. The crawler often also has options to explicitly include or exclude particular locations.

2. **Auditor**: often has options to select the types of inputs to try (form fields, URL parameters, cookies and/or HTTP headers), as well as to include or exclude specific attack vectors, such as a specific cookie name. Besides this, it has of course the option to enable and disable separate subcomponents, for instance to only scan for SQL injections. These subcomponents could have specific configuration options as well. For example, the SQLi component often has a threshold configuration option for the blind timing attacks, i.e. how long the response time should minimal be, to consider it a successful attack.

3. **Platform fingerprinter**: instead of letting the platform fingerprinter try to determine the platform, it is also possible to set it manually.

---

This approach can be taken the other way around as well, simply search for dangerous DOM-based variables and unsafe function calls. If both are found, then this is considered a (possible) DOM-based XSS vulnerability, regardless of the variable is actually used in the function call or not. This approach of course causes many false positives and thus is not used much either.
6.2. Dynamic security scanners

4. **Recon**: has similar configuration options as the auditor.

5. **Evasion**: disable and enable evasion, set maximum requests per time, set random time interval range, et cetera.

6. **Auth**: configuration options to set the authentication type (form authentication, basic HTTP authentication) and the username and password.

7. **Reports**: determine the types of reports, where to store reports, to be verbose or not, et cetera.

8. **GUI**: depends much on the scanner/GUI.

6.2.4 **Strengths and limitations**

The nature of the dynamic scanners has a number of strengths, but unfortunately also causes some limitations \[\text{Loo11, BBGM10, TS10, SP07}\]. The most important ones are the following:

**Strengths**

- Scanning is performed almost completely automated. After starting the scanner, no human intervention is needed until it is done. Furthermore, the scanner tries a huge amount of attacks and attack vectors, which would be by far infeasible to perform manually.

- Attackers often also use dynamic scanners. This is because dynamic scanners are able to scan a running web application without access to the source-code, something an attacker generally does not have. This means that if the scanners do not find any vulnerabilities, attackers also are much less likely to do so. The other way around is of course also true, if a scanner does find a vulnerability, then likely will an attacker, so it is very important to have it fixed quickly.

- The tools are perfectly suitable to analyse new products before shipping. In fact, they are perfect to test a final release candidate. However, they cannot be used to scan unfinished products.

- The scanners are language and framework independent. This results from the black-box testing approach they employ. This benefit is particularly useful for us, since it allows us to use the same analysis setup for different frameworks.

- They test the complete application stack, meaning it cannot only find vulnerabilities caused by the application, but also the ones caused by the web server, the framework, the programming language, and so on. This benefit also results from the dynamic testing approach and again this benefit is particularly useful for us, since it allows us to find vulnerabilities in the framework.

- The tools are simple to use and a scan can be started in a matter of minutes, especially for scanners with a GUI.

- Results can be reported in several convenient formats. These contain much useful information, such as a concise explanation of the attack, pointers to locations with more information and sometimes even with an example attack for the vulnerability. Some tools even include an exploit component to directly exploit the found vulnerabilities.

**Limitations**

- Dynamic scanners cannot uncover all vulnerabilities, i.e. the analysis is not complete. It is possible they miss an attack vector (location or input) during crawling. They perform black-box testing and thus cannot guarantee it covers 100% of the application and its source code.

- Even if they cover 100% of the application, they can still miss vulnerabilities. This is because, they only try some attacks, that are most likely to uncover vulnerabilities, but they do not try every thinkable attack. Therefore, there is always a possibility of a false negative result, meaning there is a vulnerability, while the scanners did not found anything. For this reason dynamic scanners are weak against competent and specially focused attackers.
Chapter 6. Options for the benchmark analysis

- It is very hard for dynamic scanners to find logical flaws, such as the usage of weak encryption or information leakage problem. For example, the scanner does not know when sensitive information is leaked. It is also hard to detect carefully hidden backdoors and multi-vector attacks: attacks that consist of chaining multiple minor vulnerabilities together, which results in a major vulnerability.

- For many types of vulnerabilities the scanners need some clues from the web application in order to detect them. For example, SQL injections are best detected if the application responds with database errors. However, if error displaying is disabled, then it is much harder to detect SQL injections. There are other techniques (blind timing attacks and blind differential analysis), but these are less reliable.

- Dynamic scanners usually have no specific goal to work toward. They simply try all attacks they know for all attack vectors they have found. This increases scan time and decreases effectiveness significantly.

- They are usually limited in the understanding of the behaviour of applications with dynamic content, such as JavaScript, Java, Flash or SilverLight. These types of content could also hinder the crawling process, further limiting the coverage of the scanners.

- Many new Web 2.0 technologies are overlooked by the dynamic scanners, while these technologies are becoming more and more popular in web applications and frameworks. Examples of technologies include: JSON, REST, HTML5, (SOAP) web services, et cetera.

- Dynamic scanners are usually slow, especially compared to static analysis tools. The main reason for this, is that they employ interactive scanning and thus have to actually try every attack. For each attack they have to perform an HTTP request, let the web server and web application process it, and wait for the response. This simply takes time. Usually the maximum number of concurrent requests is also limited, meaning the number of simultaneous attack attempts is limited as well. This causes long scan times, which is a substantial drawback to us, since the benchmark applications tend to be very large.

6.2.5 Choosing the scanners

Dynamic scanners are very popular and there are many tools we can choose, both commercial and open-source. To name a few examples:


As indicated above, the scanning process will likely be very slow. Since we do not want to waste too much time on trying dozens of different scanners, we only pick a few scanners to try further. We do not want to be solely dependent on one scanner, so we pick two. This way we can compare results and if one scanner turns out to be unsuitable, we can always continue with the other.

For selecting these two, we mostly relied on the work of others. There are scientific publications comparing dynamic scanners. However, we prefer to use an open-source scanner, for two reasons: (i) we do not have to worry about trial periods or buying licences and (ii) we can study the source-code and determine exactly how the scanners work if needed. For this reason, two papers [BBGM10, FVM07] drop out immediately, because they compared commercial tools. Furthermore, they did not publish individualized results, but anonymized them to ensure neutrality.

Two other studies remained [TS10, Loo11], that together analysed five open-source tools:

6.3. Arachni and W3af

We have chosen two specific scanners to use for the analysis: Arachni and W3af. This section further elaborates on these two scanners. First we give a short introduction describing their history and features. After that we discuss their abilities to detect the various types of SQL injection and XSS vulnerabilities. We conclude by presenting the basic configuration we used during the analysis of the benchmark for both Arachni and W3af.

6.3.1 Introduction

Arachni

Arachni is a free and open-source dynamic scanner, that was created in the summer of 2010. It is written in Ruby and has a modular architecture. It comes with a crawler and audit module, and several supportive modules, among which a platform fingerprinter and recon module. The auditor is able to scan for many vulnerability types, including (blind) SQL injections, CSRF and various types of XSS.

W3af

W3af is a web application attack and audit framework created at the end of 2006. It is developed in Python and, like Arachni, it is free and open-source. W3af also comes bundled with a GUI, but this time in the form of a (Python) desktop application. This application not only has the ability to configure and start scans and view their results, but also forms an exploit framework, which can be used to exploit found vulnerabilities. W3af’s main components include an auditor and crawler, but it also has several secondary components such as an auth module and evasion module. The auditor allows to scan for a huge number of vulnerabilities, including (blind) SQL injections and XSS vulnerabilities.

6.3.2 SQLi and XSS detection capabilities

Since our benchmark currently only includes SQLi and XSS modules, the ability to scan for these two problems is the most important for us. We will shortly describe the functionality of both Arachni and W3af regarding these vulnerability types.

Detecting SQLi

Arachni and W3af are very similar with respect to scanning for SQL injections. They both have the ability to scan for SQLi using database errors and by using blind techniques. Using error messages for detection, they support a large number of databases, including PostgreSQL. With respect to blind techniques, they both support both timing attacks as well as differential analysis for detection. The only difference is that in Arachni these can be enabled or disabled separately, while in W3af always both techniques are used.

---

For all vulnerability types supported by Arachni and other features see: [http://www.arachni-scanner.com/about/features/](http://www.arachni-scanner.com/about/features/)

For all vulnerability types supported by W3af and other features see: [http://w3af.org/plugins](http://w3af.org/plugins)
Differential analysis is not database dependent, but for timing attacks they both support only a limited amount of databases, namely MySQL, MSSQL and PostgreSQL. Fortunately, PostgreSQL is supported, which is enough for us.

Detecting standard XSS

With respect to XSS detection, Arachni and W3af differ significantly. They both support detecting standard XSS by inserting new tags in the document body. Both scanners appear to be using a combination of simple text search and document parsing to detect if there is an XSS vulnerability. This is probably for performance reasons. First a simple search is used to detect if the inserted HTML tag is present at all somewhere in the response. If not, then there is no use to parse the response. After this, the response is HTML parsed to check if the tag is actually interpreted by the web browser. If not, then we had a false positive and thus should not report the vulnerability.

Up to now W3af and Arachni operate relatively similar. The differences lie in the more advanced forms of XSS. For example, W3af offers support to detect persistent and DOM-based XSS, while Arachni does not. On the other hand, Arachni employs context-sensitive XSS detection. In the rest of this section we describe the techniques these scanners use for these features.

Detecting non-reflected XSS

W3af has the ability to detect both persistent and DOM-based XSS. With regard to persistent XSS, it employs the earlier mentioned approach to simply visit all web pages again and scan for traces of injected XSS after the auditing is finished. This approach is far from a fool-proof and is incapable of detecting all forms of persistent XSS, but it at least is a starting point. It can be enabled and disabled at will, so scan either only for reflected XSS or for both reflected and persistent XSS.

With regard to DOM-based XSS: W3af does not have an actual JavaScript runtime engine, so it only uses the simple approach using grep which we discussed earlier. This approach is not very reliable and could easily miss actual vulnerabilities, and therefore the practical use of this functionality is limited. It is included in W3af as a separate plug-in and thus can be enabled and disabled separately.

Context-sensitive XSS detection

Arachni performs context-sensitive XSS detection. It supports the standard (HTML body) context, as well as the HTML attribute context and JavaScript context. The CSS context is not supported. Recognizing three contexts sounds reasonably good, but unfortunately the implementations are rather limited. To start, Arachni simply considers everything it can inject in the JavaScript context as a vulnerability, even when it is inside a string and properly escaped. In other words, it does not recognize the JavaScript string context. This leads to many false positives, making this feature not very useful. Fortunately it can be enabled and disabled separately and Arachni properly warns about this limitation for each JavaScript result it finds.

As for the HTML attribute context, Arachni is able to detect if new unsafe attributes can be injected and this works fine. It also recognizes JavaScript event attributes, but of course with the same limitations regarding strings described earlier. The other two special cases, regarding style and URL attributes, are not supported at all. Again, the scanning for XSS in the attribute context in general as well as for event attributes specific, can be enabled and disabled at will.

W3af unfortunately does not take the context of XSS into account at all. This gives two problems: it fails to detect certain kinds of attacks and it reports many false positives, because in the current context the injected tag cannot do any damage. This is a significant drawback of W3af and therefore we have chosen not to use W3af for the XSS module, but merely focus on Arachni.

Summary

In table [6.1] we summarized the detection capabilities of both scanners. W3af’s DOM-based XSS detection is denoted with a dash, since it only uses a very basic and unreliable approach, making it impractical for real usage.
6.3. Arachni and W3af

### Detection capability

<table>
<thead>
<tr>
<th></th>
<th>Arachni</th>
<th>W3af</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQLi</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detects normal SQLi</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Supports differential analysis (blind SQLi)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Supports timing attacks (blind SQLi)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>XSS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detects standard (reflected) XSS</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Recognizes HTML attribute context</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Recognizes CSS context</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Recognizes JavaScript context</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Recognizes JavaScript string context</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Detects persistent XSS</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Detects DOM-based XSS</td>
<td>✗</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of Arachni and W3af’s detection capabilities regarding SQL injections and XSS.

#### 6.3.3 Basic configuration

This section describes the basic configuration we used during scanning for both Arachni and W3af. We also had to make some small changes to the code of Arachni/W3af in order to make them work in combination with Rails 4.0. We describe these changes, as well as the used configuration options, so it is clear which scan setup we used during the analysis.

**Arachni**

For scanning we used the newest version of Arachni available at the moment. In Arachni the user can create so-called profiles to set the scan configuration. Arachni comes with two default profiles to scan for SQL injections and XSS vulnerabilities. These profiles are very similar, the only difference being the auditor modules that are enabled. For the SQLi profile the following modules are enabled:

- Normal SQL injection (*sql*).
- Blind SQL injection using differential analysis (*sqli_blind_rdiff*).
- Blind SQL injection using timing attacks (*sqli_blind_timing*).

The XSS profile has the following modules enabled:

- Standard XSS (*xss*).
- XSS in HTML tag event attribute (*xss_event*).
- XSS in path (*xss_path*).
- XSS in HTML ‘script’ tag (*xss_script_tag*).
- XSS in HTML tag attribute (*xss_tag*).

We have created two new profiles (one for SQLi and one for XSS) based on these two default profiles, to which we made the following changes:

- **HTTP**: HTTP timeout set to 20000 milliseconds instead of 50000. This is to increase scan performance.
- **Auditor**: only audit forms is enabled; audit links and audit cookies are turned off. This is to increase scan performance as well. We can do this safely, because in our benchmark we only use HTML forms to control each functionality, as described in design choice 6.
- **Platforms**: platform fingerprinter explicitly set to operating system Linux, databases PostgreSQL and programming languages Ruby. Also to increase performance. We can safely do this because we have a fixed environment.
Plug-ins: AutoThrottle plug-in disabled. We had problems with the AutoThrottle plug-in which changed the request concurrency to very low values for no apparent reason, which made the scanning process very slow or even let it halt in the middle.

Because our SQLi scan profile is based on the default Arachni profile, we use both the normal SQLi injection auditor module, as well as the blind modules. In principle scanning only for normal SQL injections would suffice, since we enable full error reporting as discussed in SQLi design choice 4. However, to be completely sure we do not miss a potential vulnerability, we enable the blind module as well.

During our first scans with an older version of Arachni (version 0.4.5.2-0.4.2.1), we also noticed two problems in combination with Ruby on Rails applications. The first was that Arachni failed to detect the new database error format of Rails 4.0, which are PG exceptions. This problem could be solved by adding a regex rule “PG::([a-zA-Z]*)Error” to detect these to the PostgreSQL pattern file (“system/gems/gems/arachni-0.4.5.2/modules/audit/sqli/patterns/pgsql”). The second problem was a bug in Arachni’s web UI making it unable to render Rails 4.0 responses which made it impossible to view the details of a scan issue. We fixed this problem by forcing ASCII-8bit encoding for the response body (in file “system/arachni-ui-web/app/views/issues/show.html.erb”). We reported both issues to the Arachni team, and they were fixed in the next release (version 0.4.6-0.4.3).

W3af

For W3af we also used the newest version available at the time of writing. As mentioned in 6.3.2 we use W3af only for scanning for SQL injections and not for XSS, so we also only have a configuration for SQLi. This configuration is completely empty, with only the following plug-ins enabled:

- Audit: sqli and blind_sqli plug-ins with eq_limit option set to 0.9. Scan for normal as well as blind SQL injections, for the same reasons as for Arachni. The eq_limit option determines the sensitivity of the blind SQLi using differential analysis auditor. We used the default value, which turned out to work reasonably well.
- Crawl: web_spider plug-in with default settings. We need a spider plug-in to index the benchmark application.
- Output: xml_file plug-in. Unfortunately, W3af’s output plug-ins all seem to give different information. The best output is given by the GUI itself, which is only available directly after scanning. The best stored output seems to be the XML file, since it gives the most information, including the request and response data.

W3af was also not able to detect the new Rails 4 database error format. We fixed this in a similar way by adding the regex rule in Python format “(r’PG::([a-zA-Z]*)Error’, dbms.POSTGRE)” to the file “plugins/audit/sqli.py”.

74
Chapter 7

Analysing Rails with the benchmark

We have a functioning implementation of the benchmark and we have selected the tools we need for the analysis. This means, we can now actually perform this analysis, which should lead to the discovery of vulnerabilities in the Rails framework. We start by describing the analysis environment and approach. After that we present the results of this analysis, including performance results. We conclude this chapter by presenting the vulnerabilities that were found in Rails, and discuss their cause and potential solution.

7.1 Analysis environment

It is important to clearly state the environment used during the analysis, since external factors could influence the results. We first noticed this during one of our first tests, where we tried to reproduce a SQL injection in Rails’ limit\textsuperscript{1} method in order to test chosen scanners. According to the announcement this vulnerability was present in Rails 3.0.0 to 3.0.3. However, at first we were not able to reproduce it, because this vulnerability was introduced in a new version of the Arel gem, which is a dependency of Rails used to generate queries. Also, this vulnerability was not only fixed in Rails, but later also in Arel. Therefore, it was only present in Rails 3.0.0 to 3.0.3 in combination with Arel 2.0.0 to 2.0.6. This is just one example that stresses the importance of clearly describing the used environment. It is not unthinkable there are other factors that could influence the results, such as the used database, the used versions of the scanners, et cetera.

<table>
<thead>
<tr>
<th>Benchmark application</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby: 2.0.0-p0</td>
<td></td>
</tr>
<tr>
<td>Rails: 4.0.0</td>
<td></td>
</tr>
<tr>
<td>Arel: 4.0.1</td>
<td></td>
</tr>
<tr>
<td>PG: 0.17.1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL: 9.1.11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scanners</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arachni: 0.4.6-0.4.3</td>
<td></td>
</tr>
<tr>
<td>W3af: 1.6 (revision 5460aa0377)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Model: Asus M51Sn laptop</td>
<td></td>
</tr>
<tr>
<td>Processor: Core 2 Duo T9300 (2.5 GHz dual core)</td>
<td></td>
</tr>
<tr>
<td>RAM: 3 GB 667 MHz DDR2</td>
<td></td>
</tr>
<tr>
<td>Hard disk: 5400 RPM</td>
<td></td>
</tr>
<tr>
<td>OS: Ubuntu 13.10 64-bit Desktop Edition</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Overview of the analysis environment used during scanning. Arel and PG are dependencies of Rails that could potentially influence the SQLi analysis results.

\textsuperscript{1} https://groups.google.com/forum/#!topic/rubyonrails-security/tliQLPa_Tu0
The analysis environment is summarized in table 7.1. This is basically an overview of the used versions of all software. For each piece of software we used the newest version available for our operating system at the time the analysis was performed. We also list the hardware of our analysis setup, since this is useful to put the performance results into context.

To save some time we have chosen to test the benchmark only with one database system. Because Rails is largely database independent, it is likely that when Rails shows to be secure with this database, it will also be secure with other database systems. However, it is not unthinkable Rails could contain database dependent vulnerabilities. Therefore, the results we present later in this chapter are only applicable for this database and it is possible we have missed vulnerabilities that are only applicable for other databases.

### 7.2 Analysis approach

During the analysis we apply the chosen scanners on created Rails implementation of the benchmark. We perform the analysis for each module, or even submodule, separately. For example, for the SQLi module, it is recommended to separate the analysis of each submodule, as described in section 4.4.2 and 5.2.4. For each module or submodule we perform the following steps:

1. Pick and start the targeted submodule of the benchmark implementation.
2. Pick and start the scanner, either Arachni or W3af.
3. Configure the scanner correctly for the current module (as described in section 6.3.3).
4. Start the scan.
5. When the scan is finished, write down total scan duration and study the results.
6. For each result, check if it is an actual vulnerability using manual verification, i.e. try to reproduce the vulnerability in the benchmark implementation.
7. Write down the results and classification (true or false positive) with motivation and possible reason the scanner detected it.
8. For each true positive, determine whether it is a vulnerability in the benchmark application or in the framework. This is done by analysing it manually. Study the relevant code of the benchmark implementation, as well as the code of the responsible framework functionality.
9. If a framework vulnerability is found, then examine it carefully. Determine what causes it, how it can be fixed and when it was introduced. Notify the framework’s security team for responsible disclosure.
10. Repeat process for other scanners and benchmark submodules.

### 7.3 Results of the analysis

We applied the approach described above on the implementation of our benchmark in Ruby on Rails. We will now present and discuss the results of both the SQLi benchmark, as well as the XSS benchmark.

#### 7.3.1 Results of the SQLi benchmark

The SQLi benchmark was analysed with both Arachni and W3af. We will discuss these results per scanner, starting with Arachni.

**Analysis with Arachni**

The results of the analysis of the SQLi benchmark with Arachni are displayed in table 7.2. Arachni found quite some potential vulnerabilities: 55 in total. We will discuss these results per submodule, followed by a discussion of the scan run times.
7.3. Results of the analysis

<table>
<thead>
<tr>
<th></th>
<th>Normal SQLi</th>
<th>Blind SQLi diff</th>
<th>Blind SQLi time</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>TP</td>
<td>FP</td>
<td>Total</td>
</tr>
<tr>
<td>Injection</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Create</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Update</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>52</td>
</tr>
<tr>
<td>Delete</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>52</td>
</tr>
</tbody>
</table>

Table 7.2: Experimental results of the Ruby on Rails SQLi benchmark analysis using Arachni. We list the found vulnerabilities for each benchmark submodule per auditor. Total indicates the number of vulnerabilities found by that auditor. TP are the number of true positives (actual vulnerabilities), while FP indicates the number of false positives (wrong results). Time is the total running time of the complete scan for that submodule (in hh:mm:ss).

**Injection tests**

Arachni found two vulnerabilities in the injection tests module. These vulnerabilities are actual vulnerabilities, i.e. true positives. However, these were to be expected, since they were found in the injection module, which acts as our self-sanity check, as explained in benchmark design choice 9. This module contains two tests and indeed two vulnerabilities were found by Arachni, so everything seems to function correctly.

**Read tests**

There was only one vulnerability found in the read tests module. It was found by the blind SQLi using timing attacks auditor. This vulnerability is listed as a false positive result. First of all the fact that the normal SQL injection auditor did not found a vulnerability already indicates that it might be a false positive. This is because the benchmark application has full error reports enabled, thus the normal SQLi auditor should be able to detect every SQL injection. The blind auditors are only enabled as a safeguard, as was discussed in section 6.3.3. However, to be completely sure this is indeed a false positive, we employed manual verification and tried to reproduce the vulnerability. Our investigations showed the result indeed is a false positive. It might have been caused by server lag, making the server response too slow, which lead Arachni to believe its timing attack succeed.

**Update tests**

The scan found 52 vulnerabilities in the update test module. All these vulnerabilities were detected by the blind SQLi using differential analysis auditor and were categorised as false positives. Again, a first indication is that the normal SQLi auditor did not find any vulnerabilities. Furthermore, our manual verifications showed that these results are indeed false positives. They were found by Arachni using differential analysis, meaning Arachni compares responses of two attacks and if they differ too much, then there might be a vulnerability. However, these responses could also differ for many other reasons and do not necessarily have to be caused by a real vulnerability. The differential analysis was likely too sensitive, determining two responses are different even if they only differ slightly.

**Run times**

As for scan run times we see the read module takes by far the longest to execute. This is expected, since its the largest module, i.e. contains the most tests. The create, update and delete modules only take about 10-25 minutes, which is reasonable. The injection module is extremely fast, which makes sense, since it only contains two simple tests. The total execution time for the complete benchmark is below two and a half hours, which we think is reasonable, considering the old hardware the analysis was performed on. With regard to the running times we should note that these are obtained in one run only, and therefore not very reliable. This is not an issue since they are not meant for comparison, but just to give some indication of how long the analysis takes.

---

2Unfortunately, Arachni offers no option to configure the sensitivity, so we could not set it to a more logical value.
Chapter 7. Analysing Rails with the benchmark

Analysis with W3af

The results of the analysis of the SQLi benchmark with W3af are shown in Table 7.3. As can be seen from this table, W3af found only nine vulnerabilities in total, of which two were anticipated true positives and seven were unexpected false positives. Again we discuss the results per submodule.

<table>
<thead>
<tr>
<th></th>
<th>Normal SQLi</th>
<th>Blind SQLi diff</th>
<th>Blind SQLi time</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>TP</td>
<td>FP</td>
<td>Total</td>
</tr>
<tr>
<td>Injection</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Create</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Update</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Delete</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 7.3: Experimental results of the Ruby on Rails SQLi benchmark analysis using W3af.

Injection tests

W3af found two true positives in the injection tests module. Again, this was expected and indicates the benchmark and scanner are working correctly.

Update tests

The update tests module contains seven vulnerabilities according to W3af, however, all were false positives. They are similar to the ones Arachni found, and likely caused for the same reason, namely that the differential analysis was too sensitive. However, W3af found much less false positives, indicating it employs slightly less thorough analysis than Arachni.

Run times

As for the run times we see the read module takes the longest, followed by the update module. This is expected, since these two modules are the largest. The create and delete modules only take about ten minutes, while the injection module is extremely fast again. The total execution time is slightly above an hour and a half, which we think is reasonable and even better than Arachni.

Analysis of Rails 3.2

At the time we started implementing the benchmark in Rails, the current version (Rails 4.0) was not released yet. Because of this, we first implemented the SQLi benchmark in Rails 3.2. Later, we decided to switch to Rails 4. Therefore, we also have an almost complete implementation of the SQLi benchmark in Rails 3.2. We analysed this version of the benchmark as well, using the newest versions of the software available to us at that time. Therefore, the analysis environment was slightly different. We used Rails 3.2.14 in combination with Arel 3.0.2 and PG 0.17.0, PostgreSQL 9.1.9, Arachni 0.4.5.2 and W3af 1.6 (revision 3ef1aa4e9e). The results of the analysis are similar: the expected two true positives in the injection module, a number of false positives, but no real vulnerabilities. We only mention these results very briefly, without discussing them in detail, because the main focus of this research is on the 4.0 version of the Rails framework.

7.3.2 Results of the XSS benchmark

As discussed in section 6.3.2, the XSS benchmark was only analysed with Arachni. The results of this analysis are displayed in Table 7.4. As can be seen in this table, there are quite some vulnerabilities found, especially when considering the XSS benchmark implementation is far from complete. We discuss the vulnerabilities per submodule.

Injection tests

Two of the vulnerabilities were caused by the injection tests module. This is the self sanity check and
Above, we presented the results of the analysis. We have used these results to determine potential vulnerabilities in Rails and evaluate its security. In this section we will present the vulnerabilities we found.

### 7.4.1 SQLi vulnerabilities

Since all results of both scanners were either anticipated or false positives, it appears the SQLi benchmark contains no vulnerabilities and neither does the Ruby on Rails framework. This is supported by the fact that in the past seven months since the release of Rails 4, there are still no SQL injections discovered in it. We therefore believe the framework is secure regarding this vulnerability, although we can never be completely sure of this, since it is possible we missed something. We further discuss this possibility in the next chapter.

### 7.4.2 XSS vulnerabilities

We did find several XSS vulnerabilities in the simple helpers module of the XSS benchmark. In this module, Arachni detected eight true positives, of which five standard XSS and three XSS in tag vulnerabilities. We believe Arachni found these results due to three XSS vulnerabilities in the Ruby on Rails framework, which we will now explain and discuss.

---

Table 7.4: Experimental results of the Ruby on Rails XSS benchmark analysis using Arachni. The other three auditors found no vulnerabilities at all and are therefore not listed.

<table>
<thead>
<tr>
<th></th>
<th>Standard XSS</th>
<th>XSS attribute context</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total TP FP</td>
<td>Total TP FP</td>
<td>Total TP FP Time</td>
</tr>
<tr>
<td>Injection</td>
<td>1 1</td>
<td>1 1</td>
<td>2 2 0 00:07</td>
</tr>
<tr>
<td>Base</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0 0 00:19</td>
</tr>
<tr>
<td>Simple helpers</td>
<td>6 5 1</td>
<td>3 3 0</td>
<td>9 8 1 01:26</td>
</tr>
<tr>
<td>Total</td>
<td>7 6 1</td>
<td>4 4 0</td>
<td>11 10 1 01:52</td>
</tr>
</tbody>
</table>

Simple helpers tests

All other vulnerabilities were found in the simple helpers module. Of these vulnerabilities, there was one false positive which was found by the normal XSS auditor in the `cdata` helper. This helper can be used to create a CDATA section in the HTML response. CDATA sections are used to escape blocks of text containing characters which would otherwise be recognized as markup by the browser. Ruby on Rails disables the XSS protection for the content in these blocks, but it does prevent the content from breaking out of the block. However, Arachni does not detect the CDATA context and thinks it is able to place a valid HTML tag in the page, believing there is vulnerability. However, browsers ignore tags inside a CDATA section, so in fact there is no real XSS vulnerability, which makes this result a false positive.

As for the other eight results, we believe these results are true positives, indicating a vulnerability in the benchmark, which could indicate a vulnerability in the Ruby on Rails framework. After careful manual verification, we conclude there are indeed several XSS vulnerabilities in the Ruby on Rails framework present. We discuss these vulnerabilities in the next section.

Run times

As for scan run times, it is clear that the XSS benchmark is extremely fast, even on our older hardware. Of course the implementation was far from complete and scan times will increase when more modules and tests are implemented. However, giving these excellent results for the partial benchmark, we believe the scan times will be reasonable even for the full XSS benchmark.
Vulnerability in number_to_currency helper

Arachni found one standard XSS vulnerability in the `number_to_currency` helper of Rails. This helper is used to display a numeric value as a price, including a currency indicator. The currency can be set through the `unit` parameter, which caused the vulnerability. The value supplied to this parameter is not escaped, while the helper disables the automatic XSS protection. This vulnerability was also found by other people, and it was already announced (CVE-2013-6414\(^3\)) and fixed (in Rails 4.0.2) by the time we discovered it.

Vulnerability in I18n translate

We discovered one XSS in tag vulnerability with Arachni in the Rails `translate` view helper. It was actually caused by the `translate` method of the I18n module, which is used by this helper. This method is used to show a localized text for a specific translation key. However, if this key is missing, then Rails shows an error message which is wrapped in an HTML span tag, indicating which key is missing. The entered key however is not escaped, thus if a developer uses user input in the key, then this could result in XSS vulnerabilities.

At first our benchmark did not find this vulnerability. We only found it because other people discovered it and it was announced on the Rails security list (CVE-2013-4491\(^4\)). By that time, we had already implemented the translate helper in our benchmark. However, we did not find this vulnerability, because we did not test the `key` parameter. We assumed this parameter is always set by the programmer and not controlled by user input, meaning it is unsafe by design, causes false positives, and thus should be skipped. This assumption is false, since there are perfectly reasonable situations where the `key` is controlled by the user\(^5\). This implementation error is a result of the challenge of figuring out what to implement (see section 5.3.2). If Rails would clearly state which methods and parameters are unsafe by design, then we would not need to make an educated guess ourselves, which could go wrong. We discuss the problem of this challenge further in the evaluation in the next chapter.

After fixing the test for the translate helper by adding the `key` parameter, the vulnerability was detected by Arachni as expected.

Vulnerability in the simple_format helper

Arachni discovered two XSS in tag vulnerabilities in the `simple_format` helper. This helper can be used to format a text into a readable HTML text, with paragraphs and line breaks. This helper takes several options, such as the HTML tag to use for paragraphs (defaults to `p`) and the HTML attributes to set for this tag (such as `class` and `style`). However, it appears that the values for these attributes are not escaped, causing a XSS vulnerability. We tested this functionality with normal HTML attributes and special `data` attributes, since there is a special way to set these in Rails. This explains Arachni found two vulnerabilities, which were caused by the same root problem.

We reported this vulnerability to the Rails security team, who confirmed it. It was announced on December 3rd, 2013 in CVE-2013-6416\(^6\) and fixed in Rails 4.0.2. When analysing the XSS benchmark in Rails 4.0.2, Arachni did not find this vulnerability any more, indicating the fix works.

Vulnerability in the number_to_human helper

A standard XSS vulnerability was found in the `number_to_human` helper. This helper transforms a difficult to read number (very large or small number) into a nice human readable string using unit quantifiers. For example the number 1,000,000 is translated into 1 million. The helper takes an option called units, which can be used to specify names for the unit quantifiers, such that it can be used to format other decimal values, such as distances. However, the supplied values for these units are not escaped, leading to a potential XSS vulnerability, similarly to the `number_to_currency` helper vulnerability.

---

\(^3\)https://groups.google.com/forum/#!topic/rubyonrails-security/9WiRn2nhfq0
\(^4\)https://groups.google.com/forum/#!topic/rubyonrails-security/pLrh6DUw998
\(^5\)For example assume a web application where a user can select an option through a drop-down. The (internal) value is likely in the language of the programmer (often English). However, if the choice is later displayed to the user, then you would like to display the value translated. This could be implemented by calling the translate helper with the chosen option as part of the key, meaning the value is controllable by an attacker.
\(^6\)https://groups.google.com/forum/#!topic/rubyonrails-security/5ZI1-H5oILM
Vulnerabilities in format parameter of three number helpers

Arachni also detected three standard XSS vulnerabilities in several number helpers. These vulnerabilities were caused by the format option, which is supported by three of number helpers: number_to_currency, number_to_human and number_to_percentage. This option specifies the format to use for the output, i.e. where to place the number and where to place the signs, such as the unit or percent symbol. However, the value for this option is not escaped, hence causing a vulnerability. We reported this and the previous issue to the Rails security team on January 11th, 2014, who confirmed them and are currently working on a fix.

Besides the format parameter, the negative_format option of the number_to_currency helper is also vulnerable. However, the scanner did not detect this using our benchmark. The reason is that the negative_format option is not always used, but only when the helper is invoked with a negative number. It is likely that dynamic scanners do not try this and thus miss this possibility. This is a drawback of our approach using dynamic scanners, and will be discussed further in the next chapter. Helpers that alter their working based on the types of arguments should be tested in multiple ways, one for each way to invoke them. However, it is difficult to determine this in an unambiguous manner.
Chapter 8

Evaluation of the approach and results

By applying our approach on the Ruby on Rails web framework, we gained practical experience with it, as well as some insight in the security of Rails. In this section we evaluate our findings. We start by evaluating the results of the security analysis. We discuss the accuracy of these results and use this to assess the security of the Rails framework. After that, we evaluate the developed method of using a benchmark to analyse the security of web frameworks. We list general findings, describe strengths and weaknesses, and present potential improvements as well as possible other methods to analyse the security of frameworks. We conclude this chapter by presenting directions for future research.

8.1 Evaluation of the analysis results

This section discusses the accuracy of the results and gives a final assessment of the security of Rails.

8.1.1 Accuracy of the results

In chapter 7 we presented the results of the analysis of the Ruby on Rails benchmark implementation. The scanners found a reasonable amount of vulnerabilities, of which most false positives and some true positives. Some of these true positives were indeed caused by actual vulnerabilities in the Rails framework. However, most of the results were not caused by actual vulnerabilities (false positives). Furthermore, it is possible we missed vulnerabilities in the framework (false negatives) for several reasons. We discuss these reasons and explain how they influence the accuracy of the results.

False positives

False positives happen for two main reasons: (i) a flaw in the scanner or (ii) a flaw in the benchmark implementation. A flaw in the scanner occurs when the scanner indicates there is a vulnerability in the benchmark application, while there actually is none and it is impossible to reproduce this vulnerability manually. This could happen for several reasons, depending on the auditor. For example, the blind SQLi using differential analysis auditor can believe there is a vulnerability, because two responses differ too much. However, this does not necessarily have to be caused by a real vulnerability. As discussed in section 7.3.1 this happened a lot, even when we did take preventive measures as explained in section 5.2.4.

Another example of a flaw in the scanner is the false positive in the XSS benchmark simple helpers test found by Arachni. Arachni believed the cdata section helper was vulnerable only because it was unable to recognize the CDATA context properly. Usually, there is not much we can do to prevent these false positives, except wait for the developers to improve their scanners such that these false positives do not happen any more.

The second type of false positive, a flaw in the benchmark implementation, is something we can fix. These false positives happen when the scanner justly detects a vulnerability in the benchmark, but this result does not indicate a vulnerability in the framework. This could happen when the benchmark implementation uses the tested functionality wrong. An example of this would be to call a simple helper of Rails with the escape option set to false, telling Rails not to escape supplied input. Another example,

1To illustrate, when we disable the safe handle exception logic, W3af finds 1062 SQL injections and 35 blind SQL injections in the read module, whereas it otherwise finds none.
8.1. Evaluation of the analysis results

would be to implement a test for a functionality that is unsafe by design. This would be a violation of benchmark design choice 4, but it could happen by mistake. Fortunately, these are problems we can fix. However, they are usually only determined after running some first scans. If such problem is detected, then it needs to be fixed, and the scan should be run again.

We believe false positives are not that big of a problem. They are annoying, since they require manual verification, which takes extra time. However, they do not make us think that a functionality or the framework as a whole is secure, while it actually is not. Of course the number of false positives should be limited as much as possible, since they could distract the analyser from the real results. Fortunately, there are several methods to do this, such as fixing mistakes in the benchmark implementation that should not be in there in the first place, or implement inventive solutions such as the safe rescue handler. Furthermore, dynamic scanners will likely improve over time, limiting false positives even further.

False negatives

Missing a vulnerability on the other hand is much worse. Again, this occurs for two main reasons: (i) a limitation in the scanner or (ii) a limitation in the benchmark implementation. The first reason causes dynamic scanners to miss vulnerabilities, while they do exist in the benchmark implementation. Dynamic scanners are tools that help to find vulnerabilities in web applications. However, as discussed in section 6.2.4, they cannot guarantee the absence of vulnerabilities because of their design. In other words, they are not complete and the possibility exists they miss a vulnerability. Another possibility is a programming error, i.e. the scanner should detect a vulnerability, but fails to do so because of a bug. These reasons cause a false negatives: there is vulnerability in the benchmark application, but the scanner fails to detect it. An example is the inability of Arachni to detect the vulnerability in the negative_format option of the number_to_currency helper, because it does not cover the complete application (and framework) code.

With the second reason the scanners are not able to detect a framework vulnerability, simply because it is not present in the benchmark implementation at all. If the benchmark does not include a test for a particular functionality, then clearly the scanners will not find potential vulnerabilities in it either. The same holds if we do not test a functionality in all possible ways, or forget to test a particular argument or option. In other words, the accuracy of the results depends much on the completeness of the benchmark. An example of such false negative is the case of not detecting the vulnerability in the translate helper at first, because the key parameter was not tested. Basically, this false negative happened because the benchmark implementation did not fully comply with the requirement that the benchmark should test all functionalities in all the ways possible (requirement 3). It is also possible that false negatives arise due to programming errors. The benchmark does include a test for functionality containing a vulnerability, but scanners cannot detect it because the benchmark is not functioning properly. This happened once during one of our early tests. There was a bug in the safe handle exception implementation, causing it to suppress all errors. This made the normal SQLi auditor unable to detect any vulnerabilities and was the main reason we decided to include a self sanity test (requirement 10).

False negatives are a significant problem for the accuracy of our results. We want to analyse the security of a web framework and detect potential vulnerabilities in it. Missing a vulnerability could have severe consequences. Fortunately, there is much research about dynamic scanners and their accuracy keeps improving. Open-source scanners usually are reviewed by many people and often employ automated testing techniques to ensure the correct functioning. So the possibility of false negatives definitely exists (and will continue to exist), but much effort is put into work to make the probability of it as low as possible. As for false negatives due to benchmark implementation mistakes, it largely depends on the benchmark implementer. He is responsible for making sure the benchmark is implemented correctly, i.e. follows all requirements and design choices and does not contain programming errors. Unfortunately, it is very hard to verify this and it is very hard to guarantee the benchmark is implemented correctly. We do provide some ways to decrease implementation errors, for example in the form of the self sanity check, but these do not prevent everything.

Overall assessment

The overall assessment of the accuracy of the results can be divided in three parts:

83
• **True positives**: the true positives all related to a real vulnerability in the Ruby on Rails framework. In this respect the accuracy is perfect. In some cases a single vulnerability causes multiple true positive results. This could happen when the vulnerability disables the protection for multiple arguments or, in case of XSS, causes problems in multiple contexts. We do not consider this a weakness, since the single vulnerability causes multiple attack vectors, which the scanners correctly detect. Also, it is of course better to have an extra warning about a potential vulnerability then to have no warning at all.

- **False positives**: the amount of false positives are quantitative measurable. The scanner list a vulnerability, and using manual verification we can relatively easily determine if it is a false positive or not. These results are listed in section 7.3. There were some false positives, but overall the number is low. The only exception is the scan of the update module by Arachni, which gave a total of 52 false positives. This number is high, but still not insurmountable, since even for this high number we could verify quickly if the results are false positives or not.

- **False negatives**: we believe this variable is the most important, it should be as low as possible. Unfortunately, it is hard to determine this number, since we do not know how many vulnerabilities there are in Rails 4.0.0. We know of one false negative for sure, namely the missed vulnerability in the *translate* helper due to a benchmark implementation error in an early version of our benchmark. Relating this to the four other vulnerabilities we found, we believe that this is an acceptable result. However, we do not know for sure if there are not any other vulnerabilities we missed and unfortunately there is no way to find out for sure. The only thing we can do is wait and see what other vulnerabilities will be discovered.

To summarize, because of the reasonable amounts of true positives combined with the low amounts of false positives and false negatives, we believe the accuracy of our results is reasonable, definitely usable. We understand this is a bit vague, but at the time it is very hard to be more precise about this, and next to impossible to produce objective and reliable numbers.

8.1.2 The security of Rails

Using the results of the analysis and our assessment of their accuracy, we will now assess the security of the Ruby on Rails framework version 4.0.0. We analysed this framework for two types of vulnerabilities: SQL injections and XSS vulnerabilities. We did not find any SQL injections and judging from the lack of SQL injections found by others in the past seven months as well as the general low amount of SQL injections in Rails in the past few years (see figure 3.2), we believe Rails 4.0.0 is secure with respect to SQL injections. However, as explained in the previous section, we cannot be completely sure of this, because there is always a possibility of false negatives in our results.

With respect to XSS vulnerabilities, the story is quite different. In the past three years the amount of XSS vulnerabilities found in Rails decreased. However, as it appears from our results, Rails is far from secure regarding XSS. With our benchmark (final version), we detected a total of five different XSS vulnerabilities, of which one is present in three different helpers. This while this benchmark is far from complete and only tests a small part (about an eighth) of all helpers is implemented. It is likely that other helpers of Rails contain even more vulnerabilities. These findings indicate that Rails is not that secure with respect to XSS and there is significant room for improvement. However, as explained in the previous section, we cannot be completely sure of this, because there is always a possibility of false negatives in our results.

To conclude, Rails does a reasonable job with respect to security. It offers many protection mechanisms, which are often employed automatically. Regarding the protection mechanisms researched by us, as far as we can tell Rails performs excellent against SQL injections and reasonably well against XSS vulnerabilities. For the latter there is definitely room for improvement.
8.2 Evaluation of the benchmark and implementation

To start with the evaluation of the design, we believe we have specified all requirements needed to implement a benchmark to achieve the goals. These requirements are all needed, none can be removed without hindering the benchmark's goal or causing some other disadvantage. The design choices all logically relate to a requirement and are meant to ensure the benchmark as a whole complies with all requirements. Of course the design choices could also be implemented differently, while still following the requirements. However, we believe we have made the choices such that they achieve the benchmark's goals and follow the requirements, without causing to much interference, unwanted side-effects or being overly complex. Clearly, this evaluation is largely subjective and it is hard to really proof that our benchmark design is sound. We can just state that we gave the benchmark design well thought, improved it during this research and also tested in practice by applying it on the Ruby on Rails framework.

The implementation of the benchmark in Rails is already largely evaluated in section 5.4. We will shortly summarize this evaluation. Implementing the benchmark following the requirements and design choices as described in chapter 4 proved to be quite some work. We spend about a total of 150 hour working on both the SQLi and XSS benchmark together. The resulting products were relatively large Rails web applications. Furthermore the implementation required quite a deal of research in order to determine what functionalities to include, how to test these and in what ways, and what functionalities should be skipped because they are unsafe by design. There were also quite a few implementation challenges, both general ones as well as ones specific to certain tests. We believe to encounter the same problems and significant amount of work when implementing the benchmark for other frameworks.

As discussed in the previous section the implementation can also influence the results, it can cause false positives as well as false negatives. Implementing the benchmark properly and completely according to the design, turns out to be challenging. During the implementation we made several mistakes causing false negative results. Fortunately these were spotted on time and fixed before the final release, and we even were able to implement measures to prevent these in the future. However, it does show that extreme attentiveness is necessary when implementing the benchmark and all the work should be carefully checked, preferably by a separate tester. We believe that these drawbacks of the large implementation effort, large amount of required research as well as the likelihood for errors, are limitations of by the design of our approach (solution to analyse the security of web frameworks), and will be discussed in the next section.

To conclude this evaluation, we believe our benchmark design achieves its goals. Following this design, one can implement the benchmark in a framework, such that it can be used to analyse this framework for security vulnerabilities. It does so by functioning as an interface between the dynamic scanners and the framework functionalities, which cannot be tested directly. We also believe that our final benchmark implementation fully complies with the design, although we have no way to verify this. To support these conclusions we indicate that with the benchmark we were able to detect several security vulnerabilities in the Ruby on Rails framework, while undesired side-effects, such as false positives and long scan times, were kept at an acceptable rate.

8.3 Evaluation of the approach

We have evaluated the results, the benchmark design and implementation, but we did not yet evaluate the approach itself, i.e. the idea of using a benchmark in combination with dynamic scanners. In this section we evaluate our approach, discussing strengths and weaknesses. Furthermore, we look at possible other methods to analyse web frameworks and present potential improvements for our approach.

8.3.1 Assessment

We were able to implement the approach and use it to analyse the security of Ruby on Rails. During this analysis we even found several new vulnerabilities. Based on these results we could simply draw the conclusion that our approach is indeed suitable for analysing the security of web frameworks. However, this would be a bit blunt. Even though in the basis our approach is indeed capable of analysing the security of web frameworks, it unfortunately also has weaknesses, limiting its reliability and practicality. On the other hand, it also has several strengths. This section discusses these strengths and weaknesses, which are then used to draw a final assessment.
Chapter 8. Evaluation of the approach and results

Weaknesses of the approach

Our approach of using a benchmark has various weaknesses which are inheritable to the design of our solution, and cannot be easily fixed or limited. Many of these weaknesses come to expression as limitations of the benchmark, its implementation or the analysis results, and are already briefly mentioned in previous sections. However, since these limitations stem from (the nature of) our approach, we will list them here as well. We now list the most major weaknesses and the limitations that arise from them.

Benchmark implementation required for new frameworks

Although our approach is general and can in principle be used in combination with any framework, it cannot be applied directly. Instead, the benchmark must first be implemented in the specific framework. In other words it requires an instantiation phase, which is framework specific and causes extra research and implementation effort. This step is also error-prone and could influence (the reliability) of the final analysis results. Fortunately, this is the only framework specific step. Both the benchmark design and the analysis approach using dynamic scanners, are framework independent and can be reused.

False positive results

As discussed in section 8.1.1, the results of the security analysis could contain false positives. This possibility is inheritable to the choices of using dynamic scanners and of using a benchmark. We can take some measures to limit these false positives, but if not done carefully this could introduce new problems, such as false negatives. Fortunately however, the number of false positives will likely decrease over time, due to the further improvement of dynamic scanners.

False negative results

Even worse than the possibility of false positives is the possibility of false negative results. These cause us to miss vulnerabilities in the framework, and thus undermine our goal of analysing the framework’s security. As explained in section 8.1.1, there are several reasons for false negative results. These reasons are caused by our choice of using a benchmark in combination with dynamic scanners. Fortunately, we can control some of these reasons and thus limit the number of false negatives. Furthermore, dynamic scanners are always improving, limiting false negative results even further. Unfortunately however, the chance on false negatives cannot be ruled out completely, which is an inevitable disadvantage of our approach.

Not capable of testing base protection mechanisms extensively

Vulnerabilities in a framework happen mainly for two reasons: (i) the framework’s base protection mechanism fails or (ii) the protection mechanism is wrongly not applied for all cases. Our approach is not very good at detecting the first kind. Dynamic scanners try various attacks to detect vulnerabilities, so in principle they can test the base protection mechanisms. However, from what we have seen, they often only try a few basic attacks. This means that very subtle vulnerabilities in the base protection mechanisms will not be easily detected, which further increases the chance of false negatives. This is a weakness resulting from our choice to use dynamic scanners. Fortunately however, if dynamic scanners improve, then so does our approach.

Long scan times

Another drawback of our approach are the relatively long scan times. Analysing the SQLi benchmark (with Arachni) could take up to two-and-a-half hours. This is relatively long and means frameworks and new versions of it, cannot be analysed instantly, but some patience is required. This disadvantage is mainly caused due to the use of dynamic scanners, which are relatively slow. However, we believe that a scan time of a few hours is not a big deal considering it could prevent vulnerabilities which otherwise could put a lot of web applications at risk. We should also note that the scan times were obtained using relatively old and slow hardware.

Upgrading the benchmark not trivial

After the benchmark is implemented in the framework, it can be used to analyse that version of the framework. However, web frameworks are under continuous development which leads to new framework versions, that we would like to analyse as well. These developments often change the code base of the
framework and potentially its API, which could break the benchmark implementation when run in the new framework version. This means the benchmark implementation should be upgraded in order to test the new framework version, but this might not always be trivial. Especially for so-called major releases that are backwards incompatible (see section 3.2), upgrading is far from trivial and could take quite some time. For example, when we started this research we first implemented the benchmark (partially) in Rails 3.2. We later decided to switch to Rails 4.0 and upgrading the benchmark cost about sixteen man hours.

Besides from the extra effort, upgrading could also introduce implementation errors, causing false positive or false negative results. Therefore, upgrading should always be performed carefully.

Strengths of the approach

As seen above, the approach has quite some weaknesses, making it less suitable to analyse the security of web frameworks. We did however choose this approach, because it also has strengths, which we believe outweigh the weaknesses. These strengths cause several benefits that make the approach suitable to analyse the security of web frameworks, potentially more suitable than other approaches. The strengths include:

**General: applicable to any web framework**

The most important strength of our approach is that it is general. One of the main goals of this research was to find a general method to analyse web framework for security vulnerabilities. General meaning that it can be applied to any framework. Our approach is capable of that. This is mainly because of the choice to use dynamic scanners. They are general as well, and can be applied to any web framework. Unfortunately, the generality of our approach comes with a price: the approach cannot be applied instantly, since an implementation of the benchmark is required. We can however not think of any method to eliminate or even limit this instantiation phase, without compromising on the generality of our approach. We believe the possibility to apply the method to any framework is more important than the minor disadvantage of the initialisation phase.

**General: can be used with any dynamic scanner**

Not only is the approach applicable to any web framework, it can also be combined with any dynamic scanner. This gives the framework analyser options. He can choice a scanner which he thinks is reliable. If a scanner is no longer supported, it is relatively easy to switch to another. It is even possible to use multiple scanners, to get even more reliable results. Of course this would increase time needed for analysis and likely also give more false positive results, but it could be a good option to lower false negative results, which we believe is most important.

**Reusing work of others**

Our approach reuses work of others whenever possible, mostly by using already available dynamic scanners. As explained in section 4.1 this has several benefits. We have implemented the approach only for SQL injections and XSS, but this can be easily extended, since dynamic scanners can detect many other vulnerabilities as well. We can also benefit from the expertise of the makers of these scanners and the strengths of the tools that arise from that. Furthermore, once dynamic scanners improve in some way, then our approach automatically benefits from that as well.

**Relatively easy to test minor upgrades**

Although upgrading the benchmark in general is not trivial, it can be relatively easy for framework build or minor version upgrades. The build versions usually only include bug fixes, performance improvements or security fixes. They do not alter the API of the framework, and thus there is no need to alter the benchmark implementation. To illustrate this, we did not need to upgrade the benchmark at all when we analysed Rails 4.0.2 instead of Rails 4.0.0.

A minor version upgrade could also introduce new functionalities, which are of course not automatically tested and thus should be added manually. However for testing if old functionalities still work correct, upgrading is usually not required.
First step to analyse web framework security in a general manner
As mentioned in the introduction, there is not much previous work on analysing the security of web frameworks. There is some work related to the security of web frameworks, but these studies were focused on other aspects. For example, they performed qualitative research into security functionalities, such as which security functionalities do particular web frameworks implement. They did not test however if these were implemented correctly and if the framework does not contain vulnerabilities. We therefore believe that we performed a first step into analysing web frameworks for security vulnerabilities in a general manner. With our approach it is possible to search for vulnerabilities in any framework and to test how secure these frameworks really are.

Perhaps a first step towards comparing web framework security
Our approach might also be a first step towards the comparison of the security of web frameworks. Our approach searches for vulnerabilities in protection mechanisms of web frameworks. It could potentially be used to compare how well the protection mechanism of one framework works compared to another. For example, if both Rails as well as another framework B claim to supply a complete database interface that should be secure for SQL injections, then our approach can be used to verify this. For both frameworks we implement an SQLi benchmark application testing these interfaces and we use the dynamic scanners to determine the amount of vulnerabilities in them. Since both frameworks are analysed in the same way and by the same tools, we could compare the results, i.e. compare which framework contains more SQLi vulnerabilities. This is even possible if the internal workings of the protection mechanisms differ significantly. Not only can we compare numbers, we could also compare the types of attacks the protection mechanisms protect against. For example, if two frameworks both claim to supply complete XSS protection, then our approach can be used to verify if they indeed protect against all types and all contexts of XSS. It might be possible that framework B only protects in the standard XSS context, which can be detected using our approach.

Of course this is only a first step towards comparing, and great caution should be exercised when doing this. First of all, we only compare the security of protection mechanisms currently implemented in the benchmark. A framework could do a very worse job on for example CSRF protection and we will not notice this. Secondly, dynamic scanners only try certain attacks. It is possible a framework appears very secure, only because it is able to deflect the exact attacks the chosen scanners try. Thirdly, there is always a chance on false negative results arising from the benchmark implementation. Thus it is possible a framework looks very secure, only because the benchmark is not properly implemented. At last, we exclude unsafe by design functionalities, thus a framework could appear very secure, simply because it states all its functionalities are unsafe by design.

So although our approach could be a first step towards comparing the security of frameworks, there are significant caveats to keep in mind.

Assessment conclusion
With our approach we developed a general method to analyse web frameworks for security vulnerabilities. We proved this method worked by applying it to the Ruby on Rails web framework. We developed a Rails implementation of the benchmark and with it, we were able to detect a number of vulnerabilities. Unfortunately, our approach comes with several disadvantages, of which we believe the possibility of false negative results is the most severe. Unfortunately, the scanners as well as the benchmark implementation can be incomplete, causing security vulnerabilities in the framework to be overlooked. This disadvantage, as well as the other weaknesses are unfortunate. However, despite these problems, our approach is a first step into analysing the security of web frameworks in a general manner, possibly even allowing comparisons of security analyses. This is one of the strengths or our approach. We believe that this strength, combined with the others, outweighs the weaknesses and our approach is suitable to analyse the security of web frameworks and capable to detect vulnerabilities that could otherwise cause severe problems.

8.3.2 Other approaches and related work
In this thesis we introduced and discussed one approach to analyse the security of web frameworks. This is the approach we believed to be best and thus most worthy to investigate further. However, this is not the only conceivable approach. There are other possible methods as well and we discuss them in this
section. Some of them are from related work, in which the security of web frameworks is also analysed in
some way. Others are methods we thought of, but were not capable of investigating them further, either
due to time constraints or due to suspected unsolvable problems.

Qualitative/abstract analysis

Our method gives quantitative, concrete results: it find vulnerabilities in the framework (or maybe
even attacks against the framework), undermining some protection mechanism. There are however also
other approaches thinkable to analyse the security of web frameworks. These include more qualitative,
abstract approaches that analyse on a higher level. Such an approach was proposed by Weinberger et al.
in 2011 [WSA+11]. They performed what they called a systematic analysis of XSS sanitization in web
application frameworks. In their study they analysed and compared the XSS protection of fourteen major
web frameworks. They did this on a high-level, meaning they determined a list of three requirements
good XSS protection should adhere to, and then manually checked whether the web frameworks meet
these requirements. Two examples of these requirements are that the XSS sanitization should be applied
automatically and that it should be context-sensitive.

This approach could also be used to analyse the security of web frameworks. Theoretically, it could
be applied to any web framework and it could be used for other vulnerabilities that XSS as well. This
study differs from ours since it applies a qualitative and manual method. Weinberger et al. check
by hand whether frameworks comply with certain XSS sanitization criteria, which basically could be
derived from only the frameworks documentation or website. They do not check whether the framework’s
protection mechanism completely works, is fully implemented and completely achieves its claims. With
their approach they derive some general claims about the XSS protection mechanisms of web frameworks,
but they cannot discover potential vulnerabilities in these mechanisms. Our approach checks if the
framework achieves its claims and if the protection mechanism if correctly implemented. Our approach
also checks if the protection mechanism is used everywhere it is needed, which is something next to
impossible using Weinberger’s approach. They could fairly easy check if the supplied general sanitization
methods are sound (cover all subtleties of XSS), which is something our approach is less capable of [8.3.1].

However, this would require manual analysis, which requires in-depth knowledge of the vulnerability and
the framework, takes significant effort and could be error-prone and subjective.

Weinberger’s research nicely gives insight on the current standing of XSS sanitization in web frame-
works. Their method could also be a nice starting point to classify the protection mechanisms of a
framework before applying our method to find vulnerabilities in these protection mechanisms. In other
words, the approaches are complementary. Weinberger’s conclusions were that most frameworks fail to
supply proper XSS sanitization. Only seven perform automatic sanitization, of which only four apply
context-sensitive sanitization and only one recognizes all desired contexts. Furthermore, they conclude
that sanitization methods supplied by frameworks are often not fully correct and they do not protect
against certain types of XSS, such as DOM-based XSS.

API Fuzz testing

Lee et al. performed a study with a somewhat different focus [LCR08]. They tested Windows APIs for
security vulnerabilities using fuzz testing. They developed a tool that performs fuzz testing on Windows
libraries to find security vulnerabilities. However, problems in API functions that are not externally
accessible (through untrusted input) are not real vulnerabilities. To solve this issue, they implemented
a tool that finds paths between inputs of programs and faulty APIs. Where such path exists, the fault
in the API is a real vulnerability that can be exploited. To find such paths their tool used common
Windows programs, performed common actions in them and analysed if faulty APIs are called.

Although not done by Lee et al., it might be possible to apply this approach to web frameworks as well.
A web framework could be viewed as an API or library as well, it provides all kinds of functionalities
that a web developer could use. Therefore, it could be an approach to use API fuzz testing to find
vulnerabilities in web frameworks as well. However, there are functionalities that are not intended to
be exploited to unsafe input directly: the unsafe by design functionalities. Problems found in these
functionalities are not real vulnerabilities, but are false positive results. Using a similar technique as Lee
et al., it might be possible to exclude these functionalities from the results. So we use fuzz testing to
find potential vulnerabilities in a web framework and then determine whether there are paths from user
input to these functionalities. To determine the latter, we could use the same technique as Lee et al.
common web applications written in these frameworks, perform common actions in them and see if the faulty functionalities are called.

Using this approach we might be able to analyse web frameworks for security vulnerabilities. It could also limit some of the drawbacks of our approach. For example, it would not be necessary to implement a benchmark. Instead the analysis can be performed directly on the framework, using the already existing web applications to determine the accessible functionalities. Because the benchmark is eliminated it could also reduce false negative and false positive results arising from the benchmark. On the other hand however, it could increase false positive results, since faults do not necessarily have to be vulnerabilities. It could also increase false negative results, because only a very general analysis technique is employed without considering specific web application vulnerabilities. However, despite these drawbacks, it still is an interesting approach to study further.

Using static analysis tools

For our approach, we chose to use dynamic scanners instead of static analysis tools. As mentioned earlier, the reasons for this choice are the several benefits dynamic scanners have over static tools, most importantly being the generality of the first. However, static tools have a number of benefits as well, such as complete code coverage (limiting false negatives) and much smaller scan times. Furthermore, they do not need a running web application and thus could make a more direct approach possible, eliminating the need for the benchmark. This could help to remove some of the weaknesses of our current approach. For these reasons we believe static analysis tools are worthy of further investigations. However, since the options for Ruby on Rails were rather limited and outdated, we would suggest to do this for another framework.

8.3.3 Potential improvements and future research

Potential improvements

As discussed in section 8.3.1 there are several weaknesses inheritable to our approach. It would be nice to eliminate or at least limit these weaknesses, without altering the base of our approach, i.e. using a benchmark in combination with dynamic scanners. There are several possible ways to do this:

- **Reducing benchmark implementation errors**: benchmark implementation errors cause false positive or false negative results. Therefore, it would be a good strategy to reduce the chance on implementation errors and ensure the benchmark is implemented according to its design. There are several techniques to achieve this varying in degree of strictness. One approach would be to provide a check-list of common pitfalls. Another approach could be to create a formal specification of the benchmark design and use formal verification techniques to verify the benchmark implementation complies with its design.

- **Ensuring benchmark completion**: another reason for false negative results is the benchmark not being complete, i.e. it misses tests for functionalities. To prevent this we should have some way to verify the benchmark implements all framework functionalities. There are several ways to achieve this. An example would be to develop a program that creates a list of all framework functionalities that the benchmark implementer could use as check-list. Another approach would be to use tools to verify the benchmark tests cover all the code of the framework relevant for a certain type of vulnerability. For example the SQLi benchmark module should have complete code coverage of Rails’ Active Record module. It might be possible to use static analysis tools to verify this. Active Record methods not covered by the benchmark in some way are either missed functionalities or unsafe by design functionalities that should not be implemented anyway.

- **Use test generators**: instantiating the benchmark for a new framework is a labour-intensive and error-prone task. The same holds for upgrading the benchmark for future framework versions. For these reasons, it is desirable to make the implementation task easier. A way to achieve this, is by creating automatic test generators. We think this might be achievable, since the benchmark needs to test all functionalities using a standard approach: it simply needs to test all arguments and all ways the functionality can be used. Automatic test generators could also help to reduce false negative results, since it is less likely a particular functionality or way to use it is accidentally
8.3. Evaluation of the approach

excluded from the benchmark. One drawback however is that it likely increases false positive results, because unsafe by design functionalities and arguments are included as well. However, these could be removed manually later after they are discovered with the first scans. An even better way is to let the test generators analyse the framework’s documentation. This might be possible if it is machine-readable or at least has a consistent structure.

- **Make benchmark more efficient**: scanning the complete benchmark with dynamic scanners takes quite some time. It might be possible to reduce scan times by cleaning up the benchmark. For example, it could be an option to reduce the “shortcut” functionalities. The underlying methods are tested anyway, but the dynamic scanners still test them completely because they abstract from the implementation. There could also be other reasons to exclude certain functionalities, for example if it is very unlikely that it contains a vulnerability. However, these strategies are dangerous, since they increase the possibility of false negative results. We believe preventing false negative results is more important, so this should only be applied if scan times due become extremely large.

- **Improve dynamic scanners**: a very obvious way to limit the weaknesses of our approach is to improve dynamic scanners. Our approach and the results it achieves depend largely on these dynamic scanners. If we could find a way to improve them, e.g. limit the number of false positive or negative results, or improve their performance, then our methodology to analyse web frameworks for security vulnerabilities automatically improves as well.

The list above indicates potential improvements we can think of using the knowledge obtained during this research. However, these are merely hypothesis, which we did not test due to time constraints or other issues. To be sure if these approaches indeed limit the weaknesses and to what extent, further research is required.

**Future research**

We proposed a method to analyse the security for web frameworks, but as pointed out earlier, this method is not perfect yet. The first suggestion for future research is therefore to try the potential improvements listed above. Another idea for future research is to propose and evaluate other approaches to analyse the security of frameworks. Some ideas for this can be found in section 8.3.2.

It would also be interesting to test our approach on other frameworks or with other vulnerability types. Due to time constraints, we only tried the approach with Ruby on Rails. However, in order to determine general our approach really is it would be nice to implement it for a completely other framework as well. Furthermore, our benchmark currently only includes SQL injection and XSS vulnerabilities. We believe it can be easily extended with other types of vulnerabilities as well, but it would be nice to try this in practise. Furthermore, our XSS benchmark implementation is not complete. It would be nice to finish this implementation as well. Another interesting point would be to evaluate our approach better, specifically with respect to the number of false negatives. As indicated in section 8.1.1 it is hard to establish this number, because we do not know how many vulnerabilities there are in Rails 4. It could therefore be an option to implement the benchmark for an older version of Rails, for which there are many known vulnerabilities. We could then test how well the approach performs, by measuring how many of these vulnerabilities it detects. These points for further research can be used to better evaluate our approach and have the potential to discover new (non-SQLi and XSS) vulnerabilities in Rails or other frameworks.

During this research we also encountered several weaknesses and limitations of dynamic scanners. For example, dynamic scanners have the possibility on both false positive as well as false negative results. They do not cover the complete application and do not detect all kinds of possible attacks. They also are relatively slow, which is unfortunate when scanning a large benchmark application. They also have more fundamental flaws, such as their disability to detect DOM-based XSS and to recognize all XSS context properly, which was specifically a problem for W3af. Of course there are potential solutions for these problems, but these introduce new disadvantages such as further increasing false positive results or slowing down scans. Since dynamic scanners are an important tool for keeping web applications secure and can now be used to keep web frameworks secure as well, it would be nice to improve them and study them more.
Chapter 9

Conclusion

In this thesis we proposed, implemented and evaluated a general approach to analyse the security of web frameworks. With this approach it is possible to detect security vulnerabilities in arbitrary web frameworks. The approach uses a benchmark in combination with dynamic scanners. By using a benchmark we can overcome some of the challenges when analysing frameworks, as discussed in section 4.1. By using the dynamic scanners we can reuse the work of others and partially automate our approach. There are other options for the analysis, but as discussed in chapter 6 dynamic scanners are the most suitable for our needs. We designed the benchmark and two of its modules: the SQLi module and XSS module. This design is one of the main products of this research. We have chosen for SQLi and XSS since they are severe and very common in web applications.

Before the benchmark can be used, it needs to be implemented in the framework. This can be seen as an instantiation of the benchmark for that framework. It results in a web application for each benchmark module. We did this for the Ruby on Rails framework and this implementation is another product of this research. In this thesis we discussed the implementation in Rails, including various implementation challenges we came across.

Using our benchmark implementation, we were able to analyse Rails for security vulnerabilities. We performed the analysis using two dynamic scanners: Arachni and W3af. The majority of the results were false positives and we did not discover any SQL injection vulnerabilities. However, we did find several XSS vulnerabilities that were confirmed by the Rails security team and will be fixed soon. These XSS vulnerabilities threatened Rails web applications and could have severe consequences. Attackers could use them to hijack sessions, display false content on websites, steal sensitive cookie data and more. Thanks to our approach they were detected relatively soon and fixed before any real harm was done.

By performing the security analysis of Rails, we proved that our approach is indeed capable of analysing web frameworks for security vulnerabilities. Therefore, we can answer our research question positively. However, there are some caveats. The approach has some weaknesses that are inheritable to its design. The most important are the large instantiation effort required for new frameworks and the possibility of false negative results. Some of these weaknesses are significant, but we believe the strengths outweigh them and think our approach is a good first attempt to analyse the security of web frameworks in a general manner. We also present some potential improvements, that might reduce these weaknesses, but this is subject to future research.
Bibliography


Appendix A

Implemented Rails functionalities

This chapter presents an overview of all implemented Rails functionalities per benchmark module. We also list the methods deliberately excluded and the reason for that.

A.1 Implemented SQLi benchmark functionalities

Implemented functionalities

The Rails SQLi benchmark related functionalities are all implemented as methods of the Active Record component. They come in the form of three types: object methods, class methods and relation methods (see section 5.2.2). The following methods are implemented in the benchmark:

<table>
<thead>
<tr>
<th>Create module</th>
<th>Read module</th>
<th>Update module</th>
<th>Delete module</th>
</tr>
</thead>
<tbody>
<tr>
<td>class.create</td>
<td>relation.create_with</td>
<td>relation.update</td>
<td>relation.delete</td>
</tr>
<tr>
<td>class.create!</td>
<td>relation.eager_load¹</td>
<td>relation.update_all</td>
<td>relation.delete_all</td>
</tr>
<tr>
<td>relation.create</td>
<td>relation.includes¹</td>
<td>object.save</td>
<td>relation.destroy</td>
</tr>
<tr>
<td>relation.create!</td>
<td>relation.joins¹</td>
<td>object.save!</td>
<td>relation.destroy_all</td>
</tr>
<tr>
<td>object.save</td>
<td>relation.limit</td>
<td>object.update</td>
<td>object.delete</td>
</tr>
<tr>
<td>object.save</td>
<td>relation.offset</td>
<td>object.update!</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relationpreload¹</td>
<td>object.update_attribute</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relation.distinct</td>
<td>object.update_attributes</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relation.where</td>
<td>object.update_attributes!</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relation.any?</td>
<td>object.update_column</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relation.blank?</td>
<td>object.update_columns</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relation.empty?</td>
<td>object.increment!</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relation.find_or_initialize_by</td>
<td>object.decrement!</td>
<td>object.destroy</td>
</tr>
<tr>
<td></td>
<td>relation.find_or_create_by</td>
<td>object.toggle!</td>
<td>object.touch</td>
</tr>
<tr>
<td></td>
<td>relation.find_or_create_by!</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.first_or_initialize</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.first_or_create</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.first_or_create!</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.many?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.to_a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.explain</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.inspect</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.load</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.reload</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.all</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.exists?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relation.find</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ These methods include eager loading for performance optimization.
A.1. Implemented SQLi benchmark functionalities

<table>
<thead>
<tr>
<th>Relation Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>relation.find_by</td>
<td></td>
</tr>
<tr>
<td>relation.find_by!</td>
<td></td>
</tr>
<tr>
<td>relation.first</td>
<td></td>
</tr>
<tr>
<td>relation.first!</td>
<td></td>
</tr>
<tr>
<td>relation.last</td>
<td></td>
</tr>
<tr>
<td>relation.last!</td>
<td></td>
</tr>
<tr>
<td>relation.take</td>
<td></td>
</tr>
<tr>
<td>relation.take!</td>
<td></td>
</tr>
<tr>
<td>relation.find_by_&lt;attributes&gt;²</td>
<td></td>
</tr>
<tr>
<td>relation.find_by_&lt;attributes&gt;!²</td>
<td></td>
</tr>
<tr>
<td>relation.average</td>
<td></td>
</tr>
<tr>
<td>relation.calculate</td>
<td></td>
</tr>
<tr>
<td>relation.count</td>
<td></td>
</tr>
<tr>
<td>relation.ids</td>
<td></td>
</tr>
<tr>
<td>relation.maximum</td>
<td></td>
</tr>
<tr>
<td>relation.minimum</td>
<td></td>
</tr>
<tr>
<td>relation.pluck</td>
<td></td>
</tr>
<tr>
<td>relation.sum</td>
<td></td>
</tr>
<tr>
<td>relation.find_each</td>
<td></td>
</tr>
<tr>
<td>relation.find_in_batches</td>
<td></td>
</tr>
<tr>
<td>relation.select³</td>
<td></td>
</tr>
<tr>
<td>relation.find_by_sql</td>
<td></td>
</tr>
<tr>
<td>relation.count_by_sql</td>
<td></td>
</tr>
</tbody>
</table>

1. Only implemented when called with Symbol/String, not with Array/Hash (to load multiple associations).

2. Only for a single attribute, not for a dynamic list of multiple attributes.

3. Only when called with a block (it then executes the query), not when called with a string (sets the SELECT part of the query), since this is unsafe by design.

Some methods can be used in multiple ways, depending on their arguments and the types of these arguments. An example is the `relation.exists?` method, which can be called with no arguments, with a single ID (integer) as argument, or with an array or hash representing conditions. In principle all methods are tested in all ways possible, unless clearly stated otherwise. This sometimes leads to separate web pages in the benchmark.

### Implemented condition forms

The query methods `relation.where` and `relation.having` take conditions as input. Rails has various ways to set the conditions, which influence the final query. These methods therefore take various argument types:

1. **String**: the argument is an SQL fragment that is directly used in the query (no sanitization applied).
   This type is included in the benchmark, and manual sanitization is applied, which also immediately tests a few of Rails’ base sanitization methods.

2. **List**: the method is called with a list of arguments, where the first is an SQL fragment with placeholders (which is not sanitized) and the rest are values for these placeholders (which are automatically sanitized). This type is also included in the benchmark. There are various styles for the placeholders, which are all included:
Appendix A. Implemented Rails functionalities

3. **Array**: same as list, only now the method is called with a single array (containing the list). Also implemented in the benchmark.

4. **Hash**: the method is called with a hash. This hash is translated to conditions. The keys of the hash correspond with database columns, for which an equality condition is created on the value for that key. The values are automatically sanitized. This method is implemented in the benchmark. Several data types are supported:

   a) ActiveRecord::Relation: the value is a relation object (representing another query). This will result in a subset condition on the returned values by this query, e.g. `column IN (<list of IDs returned by other query>).` This method is not implemented, since it is not exploitable by direct user input.

   b) ActiveRecord::Base: the value is an Active Record object (Rails model). This will result in an equality condition on the object’s primary key (ID), e.g. `column = 3` if the object’s ID was 3. This method is not implemented, since it is not exploitable by direct user input.

   c) Class: the value is a Ruby class. This will result in an equality condition on the class’ name, e.g. `column = 'Book'`. This method is not implemented, since it is not exploitable by direct user input (and a comment in the code says this behaviour should be deprecated).

   d) Array: the value is an array (e.g. `[1,2,4,5]`). This results in a subset condition on the values in the array, e.g. `column IN (1,2,4,5)`. This is implemented in the benchmark.

   e) Range: the value is a range (e.g. `1..5`). This results in a range condition, e.g. `column BETWEEN 1 AND 5`.

   f) **Other types**: the value is something else, for example a string "hi" or an integer 3. This will result in an equality condition on the values string representation (obtained by calling the `to_s` method), e.g. `column = 'hi'` or `column = 3`.

5. **Other types**: the method is called with an argument of some other type. For most types this results in a type error or SQL error. It can probably be used to extend Active Record to allow conditions of some custom type. This method is not included in the benchmark, since in most cases it does not work and we do not believe it is very common in Rails applications.

**Excluded functionalities**

Some of Rails’ Active Record methods were deliberately not implemented for one of these reasons:

- **Unsafe by design methods**: some methods are unsafe by design, i.e. their values are directly included in the query, without applying sanitization. These methods include: `relation.from, relation.group, relation.lock, relation.order, relation.reorder, relation.select` (when called with a string). All methods are related to the read module.

- **Not applicable methods**: some methods are not applicable, because they do not affect or execute the query. An example of such method is `relation.readonly`, which sets a flag on the found objects that they cannot be altered.

- **Skipped methods**: some methods do alter the final query, but are skipped anyway. These methods include the three SpawnMethods (read module): `relation.except, relation.merge` and `relation.only`. These alter the values for the query methods, such as the value for the LIMIT part of the query. However, they do not overwrite values directly, but merely choose which values to include in the final query. Therefore, they are no entry point for unsafe user input and thus can be skipped.
A.2. Implemented XSS benchmark functionalities

Implemented functionalities

The Rails XSS benchmark related functionalities are almost all implemented as view helpers in the Action View component. The following helpers are implemented in the benchmark:

<table>
<thead>
<tr>
<th>Base module</th>
<th>Normal helpers module</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic protection</td>
<td>number_to_currency</td>
</tr>
<tr>
<td>sanitize</td>
<td>number_to_human</td>
</tr>
<tr>
<td>sanitize_css</td>
<td>number_to_human_size</td>
</tr>
<tr>
<td>strip_tags</td>
<td>number_to_percentage</td>
</tr>
<tr>
<td>escape_once</td>
<td>number_to_phone</td>
</tr>
<tr>
<td>escape_javascript</td>
<td>number_with_delimiter</td>
</tr>
<tr>
<td>html_escape</td>
<td>number_with_precision</td>
</tr>
<tr>
<td>html_escape_once</td>
<td>cdata_section</td>
</tr>
<tr>
<td>json_escape</td>
<td>content_tag</td>
</tr>
<tr>
<td></td>
<td>tag</td>
</tr>
<tr>
<td></td>
<td>highlight</td>
</tr>
<tr>
<td></td>
<td>simple_format</td>
</tr>
<tr>
<td></td>
<td>truncate</td>
</tr>
<tr>
<td></td>
<td>translate</td>
</tr>
<tr>
<td></td>
<td>button_to</td>
</tr>
<tr>
<td></td>
<td>link_to</td>
</tr>
<tr>
<td></td>
<td>link_to_if</td>
</tr>
<tr>
<td></td>
<td>link_to_unless</td>
</tr>
<tr>
<td></td>
<td>mail_to</td>
</tr>
</tbody>
</table>

1 This method takes a whitelist of allowed tags and attributes. We tested this method in various configurations: with no tags and attributes allowed, with no tags allowed, with no attributes allowed, with a custom list of tags and attributes allowed and with the default tags and attributes allowed.

2 This method usually returns a non-HTML-safe value, which means automatic protection is also applied. This is done because these methods perform less strict sanitizing, possibly allowing some very specific vulnerabilities. Therefore, developers should consciously disable the automatic protection when using these methods. We test these methods with both the automatic protection enabled and disabled (by calling html_safe on the return value).

Again, some methods can be used in multiple ways, depending on their arguments and the types of these arguments. An example is the number_to_human helper which can be called with a hash containing the units, as well as a string pointing to a translation key containing the units. In principle all helpers are tested in all ways possible (which sometimes leads to separate web page per way). Many view helpers be called with a block, which is interpreted and then used in the return value. Calling a helper with a block often leads to (slightly) different behaviour. We have tested all methods that accept blocks both with and without a block.
Excluded functionalities

Some of Rails’ view helpers were not implemented in the benchmark for one of the following reasons:

- **Unsafe by design helpers**: some helpers are unsafe by design, i.e. their values are directly included in web response, without applying sanitization. The most obvious example is the `raw` helper, which disables automatic protection by calling `html_safe` on the input string. Another example is the `javascript_tag` that can be used to create an HTML script tag with content. Obviously this method is unsafe by design, since it wraps the content in a script tag (and disabled automatic protection), allowing XSS attacks.

- **Not applicable helpers**: helpers methods are not applicable, because they are not meant to output content on the response page. An example of such helper is the `current_page?` helper which can be used to check if a page number matches the number of the current page. This method return a boolean, which is meant to be used in conditional statements, rather than being outputted on the page directly. Clearly this method would also not cause an XSS vulnerability, since it does not output unsafe user input.

- **Deprecated helpers**: some helpers are (silently) deprecated. We chose to exclude these, since they will be removed in future versions of the framework anyway. An example is the `button_to_function` helper, which is silently deprecated in favour of unobtrusive JavaScript [Fla06].

- **Not yet implemented helpers**: many helpers are simply not yet implemented while they should be. The XSS benchmark implementation unfortunately was not finished due to time constraints. This does not mean however that these helpers should not be implemented in the final version.
Appendix B

Safe rescue exceptions

As discussed in section [5.2.4] we use a safe rescue exception handler to suppress various exceptions. This is necessary because the exception could be misinterpreted by the dynamic scanners as potential SQL injections, causing false positives. Since dynamic scanners simply try any kinds of values (they have no knowledge of the underlying database field or its type), they could trigger these exceptions. For example, the scanner might try to update the integer column of some record with an invalid value, such as `aaa`. If this happens, the PostgreSQL query will fail and an exception is triggered. This could trick the scanner into believing there is an SQL injection vulnerability, while there actually is none. The safe rescue exception handler prevents this by intercepting and suppressing the exception and trying to render the default response instead. It uses a whitelist of exception types and (parts of) messages to determine which exceptions need to be suppressed. The exceptions included in this list for the SQLi benchmark are:

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Message</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG::UniqueViolation</td>
<td>ERROR: duplicate key value violates unique constraint</td>
<td>Happens when primary key (id) is overwritten, for example by setting it through the <code>create_with</code> method</td>
</tr>
<tr>
<td>PG::InvalidParameterValue</td>
<td>ERROR: invalid hexadecimal digit</td>
<td>Happens when setting binary_col with invalid values (usually when bypassing the Active Record object, for example with <code>update_column</code> method)</td>
</tr>
<tr>
<td>PG::InvalidTextRepresentation</td>
<td>ERROR: invalid input syntax for type</td>
<td>Happens when setting invalid values for columns with special types, such as booleans and decimals (usually when bypassing the Active Record object)</td>
</tr>
<tr>
<td>PG::InvalidTextRepresentation</td>
<td>ERROR: invalid input syntax for integer</td>
<td>Happens when setting invalid values for integer columns (usually when bypassing the Active Record object)</td>
</tr>
<tr>
<td>PG::InvalidDatetimeFormat</td>
<td>ERROR: invalid input syntax for type</td>
<td>Happens when setting invalid values for datetime columns (usually when bypassing the Active Record object)</td>
</tr>
<tr>
<td>PG::DatetimeFieldOverflow</td>
<td>ERROR: date/time field value out of range</td>
<td>Happens when setting out of range values for datetime columns, for example the value 0 (usually when bypassing the Active Record object)</td>
</tr>
<tr>
<td>Exception Type</td>
<td>Error Message</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PG::SyntaxError</td>
<td>ERROR: syntax error at or near &quot;DISTINCT&quot; ... DISTINCT DISTINCT</td>
<td>Happens due to a bug in Rails' query methods when setting a value for the <code>limit</code> method, setting the <code>distinct</code> method on true and setting the <code>eager_load</code> method on <code>has_many</code> or <code>has_and_belongs_to_many</code>.</td>
</tr>
<tr>
<td>PG::AmbiguousColumn</td>
<td>ERROR: column reference “id” is ambiguous</td>
<td>Happens when setting a value for query method <code>having</code> while also setting a value for <code>eager_load</code> or <code>joins</code> (it probably expects a different input format for <code>having</code> then).</td>
</tr>
<tr>
<td>ActiveRecord::RecordNotFound</td>
<td></td>
<td>Happens when we try to find a non-existing record, for example by using the <code>find</code> method with an unknown id.</td>
</tr>
<tr>
<td>ActiveRecord::ConfigurationError</td>
<td>Association named ... was not found</td>
<td>Happens when trying to set an association method (for example <code>includes</code>) with an invalid association name.</td>
</tr>
<tr>
<td>ArgumentError</td>
<td>invalid value for Integer()</td>
<td>Happens when setting an invalid value for the <code>limit</code> query method.</td>
</tr>
<tr>
<td>ArgumentError</td>
<td>argument out of range</td>
<td>Happens when setting a very small or very large value (such as 0000-00-00 00:00:00) for a datetime field (and possibly other fields).</td>
</tr>
</tbody>
</table>