MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Information Management and IT Security

SCRAP - Secure Code Review Automated Platform. Prototype of an automated feedback platform to provide secure coding feedback for introductory programming courses.

By: Andrea Ida Malkah Klaura, MA BSc

Student Number: 1710303005

Supervisors: Dipl.-Ing.(FH) Mag. DI Christian Kaufmann
Ing. Bernhard Lueger MSc

Wien, April 28, 2020

FH University of Applied Sciences TECHNIKUM WIEN

Declaration

Kurzfassung

Eine der größten Schwächen der Cybersicherheit liegt in der “menschlichen Infrastruktur” (Hadnagy, 2011, 3). Während dies in der Regel zu Security Awareness Trainings der MitarbeiterInnen führt und andere organisatorische Fragen aufwirft, wird das Bewusstsein über sicheren Code unter den EntwicklerInnen zu oft vernachlässigt. Ein wichtiger Hebel zur Verbesserung der Informationssicherheit ist die Verbesserung der Qualität des Codes in Bezug auf sichere Programmierung.

Ausgehend von dieser Situation untersucht meine Masterarbeit die Machbarkeit der Verwendung von Freien und Open-Source-Software (F/LOSS) Tools, um eine Toolchain und Plattform zur Generierung von Feedback aufzubauen, die für einführende Programmierkurse verwendet werden könnte, um Anreize für die Sensibilisierung und Praktizierung sicheren Programmierens zu schaffen. Eine Auswertung von 7 von 19 gefundenen statischen F/LOSS Analysewerkzeugen für PHP-Code, zeigt, dass nur 2 zum Teil für eine Analyse hinsichtlich Sicherheitsschwachstellen im Code geeignet sind. Mehrere dieser Tools bieten jedoch Möglichkeiten zur Erweiterung und Adoption. Die Verwendung von 2 ausgewählten Werkzeugen im SCRAP-Prototyp, bietet einen Ausgangspunkt für weitere potentielle Forschung.

Der SCRAP-Prototyp, der im Zuge dieser Arbeit entwickelt wurde, besteht aus einer OpenAPI 3 konformen API, einer entsprechende Prototyp-Implementierung eines RESTful Web Services und einem prototypischen Web UI, die auf der Projekt-Website https://scrap.tantemalkah.at dokumentiert, und unter einer AGPLv3-Lizenz zugänglich sind. Die Implementierung ist einfach mit zusätzlichen Scannern erweiterbar und kann für weitere Forschung im Bereich der *software security education*.

Während die umfangreiche Literaturanalyse zeigt, dass der Bereich der *software security education* noch viel mehr Aufmerksamkeit braucht, fügt SCRAP hier einen neuen Ansatz hinzu. Seine Umsetzung erfordert jedoch einen nachhaltigen, langfristigen Ansatz und soziotechnische Adaptionen in Organisationen, die diesen Ansatz erfolgreich umsetzen wollen, sowie mehr offene und kooperative Forschung im Bereich der *software security education*, um Verbesserungen bei der sicheren Programmierung im gesamten Bildungssektor und im Weiteren in der IT-Branche im Allgemeinen zu erzielen.

Schlagworte: software security education, secure code, static code analysis, F/LOSS toolchain

Abstract

One of the biggest weaknesses in cybersecurity lies within the “human infrastructure” (Hadnagy, 2011, 3). While this usually leads to security awareness trainings of employees and other organisational issues, the awareness for secure code among developers is too often neglected. A major lever to improve information security is to improve the quality of code in terms of secure coding.

Based on this situation, my thesis investigates the feasibility of using Free/Libre and Open Source Software (F/LOSS) tools, to build a toolchain and feedback generation platform, that could be used in introductory programming courses to add incentives for secure coding awareness and adoption. An evaluation of 7 out of 19 found F/LOSS static analysis tools for PHP code analysis, shows that only 2 are in part fit for secure coding specific analysis. However, several of those tools provide opportunities for extension and adaptation. The use of 2 selected tools in the SCRAP prototype provides a starting point for further potential research.

The SCRAP prototype, that was built in the course of this thesis, consists of an OpenAPI 3 conforming API, a corresponding prototype implementation of a RESTful web service and a prototype web UI, which are documented on the project website https://scrap.tantemalkah.at, and are accessible under an AGPLv3 license. It is easily extendible with additional scanners and can be used for further research into software security education.

While the extensive literature review reveals that the field of software security education still needs a lot more attention, SCRAP adds a new approach. Yet its adoption necessitates a sustainable long-term approach and socio-technical adaptations in organisations who want to facilitate it, as well as more open and cooperative research in software security education is needed to improve secure coding capabilities throughout the educational sector and in term the IT industry in general.

Keywords: software security education, secure code, static code analysis, F/LOSS toolchain

Contents

1 Intro

One of the biggest weaknesses in cybersecurity landscapes lies in the “human infrastructure” (Hadnagy, 2011, 3). No matter how complex and thorough an organisation deploys technical safeguards against cyber attacks, if they do not mitigate social engineering attacks, chances are high that targeted attacks will succeed (Hadnagy, 2011, 1-21). Security awareness is therefore one of the keys to secure information systems (Helisch & Pokoyski, 2009).

Besides the usually broader organisational issue of security awareness, there is another crucial issue entangling cybersecurity and awareness: the awareness among software developers about code vulnerabilities and how code can be written in a secure way. According to a late 2018 article by Code Dx, referencing the F5 Labs Application Protection Report 2018 (Pompon, 2018), “[w]eb application attacks are on the rise [and] were the primary cause of reported breaches in 2017 and Q1 2018” (Code Dx, 2018). They also highlight that a “[l]ack of attention to security is also an issue” resulting in a situation that “96 percent of all web applications contain some type of vulnerability that could be used to harm users.” For these numbers they reference the *Web Application Vulnerabilities Statistics* report for 2017 by Positive Technologies (Positive Technologies, 2018). This highlights the necessity for developers to be aware about vulnerabilities in web applications and secure coding in general. In a more recent Positive Technologies report exposes two most alarming core findings:

- “In 19 percent of tested web applications, vulnerabilities allow an attacker to take control of the application and server OS” (Positive Technologies, 2019*b*)
- “On average, each web application contained 33 vulnerabilities, of which 6 were of high severity” (ibid)

Another recent Positive Technology report containing statistics on pentesting corporate information systems found that “[v]ulnerabilities in web application code are the main problem on the network perimeter. 75 percent of penetration vectors are caused by poor protection of web resources.” (Positive Technologies, 2019*a*)

When we look beyond web application security, to the general software landscape, the recent volume 9 of the Veracode report on the State of Software Security highlights that “[m]ore than 85% of all applications have at least one vulnerability in them; more than 13% have at least one critical severity flaw.” (Veracode, 2018).

In light of this background it seems quite clear that, besides increasing security awareness on all organisational and societal levels, a major lever to improve information security is to improve the quality of code in terms of security. As several actors in the secure coding research field argue (Teto et al., 2017) (Jøsang et al., 2015) (Raina et al., 2015) (van Niekerk & Futcher, 2015), an important and still under-represented step in doing so is to integrate secure coding into programming education. While we already see major frameworks that include security issues into the whole software development life cycle, there is still too little time and space set aside for secure code in technical college and university introductory programming courses.

These introductory programming courses are usually crammed with all the things students should know to be able to solve any given standard exercise. And all of this is often put into one or two 5 to 10 ECTS credit courses. As an example, we can take a look at two local universities in Vienna, both with technology focus, including programmes and courses on IT security.

At the FH Technikum Wien there is a bachelor degree in computer science, which includes a 4.5 ECTS introductory course on programming in the first semester, as well as a 3.0 ECTS course on web technologies. Both two courses have a mandatory follow-up course with the same ECTS amount in the second term (FH Technikum Wien, 2019).

At the TU Wien there are several different computer science bachelor programmes, with the *BSc Media Informatics and Visual Computing* and the *BSc Software and Information Engineering* being the closest fit for web application developers and developers coding in PHP, which is still by far the most used server side programming language in web applications, with a market share of over 78% (W3Techs, 2020). Both programmes include two consecutive introductory programming courses with 5.5 and 4.0 ECTS respectively in the first and second term, as well as an optional 6.0 ECTS module on development of web applications (with 3 ECTS on semi-structured data and 3 ECTS on web engineering). The software information and engineering programme further includes a mandatory 6.0 ECTS course on programming paradigms (Technische Universität Wien, 2019*a*) (Technische Universität Wien, 2019*b*). Both also include an optional 6.0 ECTS module consisting of several 3.0 ECTS courses to choose from, none of which has a specific focus on secure coding [1].

We can assume, that these two cases are not an exception in the global computer science education landscape. The following literature review corroborates this assumption.

To get back to the point: in a relatively short time programming students are supposed to learn to implement a broad range of functional requirements. Usually the set of requirements to be implemented does not contain security requirements. This is not different from general software engineering contexts. As Conklin et al. put it:

> “Security has been described as a nonfunctional requirement. This places it into a category of secondary importance for many developers. Their view is that if timelines, schedules, and budgets are all in the green, then maybe there will be time to

[1]One of these courses, *Security for Systems Engineering* contains the topics *Security in Software Development* and *Web Application Security*, but these are 2 of the 11 main course topics, ranging very broadly from *Cryptography* and *Network Security* to *Organizational Security* and *Risk Management*

devote to security programming." (Conklin et al., 2016, Ch. 18, Para. 2)

Based on this situation, my thesis investigates the current state of software security education and the feasibility of using Free/Libre and Open Source Software (F/LOSS) tools, to build a toolchain and feedback generation platform, that could be used in introductory programming courses to add opportunities for secure code awareness, without overloading the existing coursework.

Chapter 2 highlights the results of my literature review in four parts: section 2.1 provides an overview of current standards of code security and my analytical frame to focus on code and its potential vulnerabilities. Section 2.2 highlights some current platforms and tools that are available for static code analysis. Section 2.3 provides insights into the field of software security education and efforts to include secure coding and security awareness into college and university programming courses. In section 2.4 I point towards other research that aims to include some form of educational code analysis tools or toolchains into programming education. Following the analysis of the current state of the field, I formulate my research question in chapter 3. In the methods chapter (4), I provide detailed insights into how I conducted the literature review (4.1), in order to identify the huge gaps, which we still have to fill in software security education and also the educational technologies available to improve the secure coding awareness and skills of developers. I also provide some methodical insight into my prototyping approach (4.2). Chapter 5 focuses on the prototype that was developed in this project, and provides details about the design, requirements and architecture (5.1), the overall implementation (5.2), and its evaluation (5.3). In the penultimate chapter (6) I point out further directions that could be taken in follow-up research. The concluding chapter 7 summarizes the results and points towards the applicability of F/LOSS tools to be integrated into higher education programming courses in order to improve code security.

2 Current State

2.1 Secure Coding

The human factor is probably the biggest issue in cyber security (Hadnagy, 2011) (Helisch & Pokoyski, 2009). Although this usually refers to the users of information systems, when it comes to their implementation most vulnerabilities can be traced back to bad programming (Stallings & Brown, 2018) (OWASP, 2019*b*) (MITRE & SANS, 2011). This chapter points out the general importance of writing secure code as well as reference frameworks and perspectives on how to tackle secure code and code vulnerabilities from an analytical point.

In *Principles of Computer Security*, the official CompTIA guide, three major considerations are mentioned for securing a digital information system:

- Correctness
- Isolation
- Obfuscation

While for the latter two the authors primarily focus on network design and cryptography, for the issue of *correctness* they state:

> "Correctness begins with a secure development lifecycle (covered in Chapter 18), continues through patching and hardening (Chapters 14 and 21), and culminates in operations (Chapters 3, 4, 19, and 20)." (Conklin et al., 2016, Ch. 1, Section: Approaches to Computer Security)

So even in this standard reference guide on computer security, the topic of secure code is reduced to a one chapter item in the whole security framework. That also has to do with what cyber security frameworks mostly are here for: to help organisations, which mostly use information & and communication technologies (ICTs) and not so much develop them on their own, to strengthen the secure facilitation of these ICTs on all levels - hence the chapters on patching and hardening and all the guidelines on how to operate ICT securely.

Similarly international standards on information security management like ISO 27001 are used for the certification of organisations operating with high levels of information security (ISO, 2013). They can and are of course also be used for organisations developing code. But from their design they are aimed to be implemented for a most diverse set of organisations, which means it is primarily aimed towards organisations that have to depend on using code by others rather than developing their own code.

But while such information and computer security reference books like *Principles of Computer Security* (PoCS) do not focus primarily on secure code, we can nevertheless find some important aspects either relating or directly pertaining to secure coding. Chapter 2 of PoCS, for example, introduces three main security tenets (Conklin et al., 2016, Ch. 2, Section: Security Tenets) and refers to the OWASP Session Management Cheat Sheet (OWASP, 2019*c*), which guides developers on how to securely implement session management in their applications. It also is a concrete but language-agnostic guide on what to consider when implementing session management.

The other mentioned tenets besides “Session Management” are “Exception Management” and “Configuration Management”. While exception handling in a software is part of *exception management*, the focus here lies on the general working of the whole information system and its security processes. *Configuration management* is mostly out of scope of secure coding.

The chapter focusing on the secure development life cycle (Conklin et al., 2016, Ch. 18, Section: Secure Coding Concepts) lists 4 concepts applied for secure coding:

- Error and Exception Handling
- Input and Output Validation
- Fuzzing
- Bug Tracking

While fuzzing and bug tracking are important in testing and managing the software development, for our purposes we have to consider error/exception handling and I/O validation as a main focus. Besides these concepts for secure coding they also list concrete types of attacks, which secure code should mitigate (Conklin et al., 2016, Ch. 18, Section: Application Attacks):

- Cross-Site Scripting
- Injections (specifically with SQL, LDAP and XML)
- Directory Traversal/Command Injection
- Buffer Overflow
- Integer Overflow
- Cross-Site Request Forgery
- Attachments
- Locally Shared Objects
- Client-Side Attacks

- Arbitrary/Remote Code Execution

Preceding the chapter on secure software development, they provide a whole chapter on web components. There we find general background information on security concerns regarding the different technologies and protocol layers surrounding web usage. When it comes to code security for web applications they refer to the chapter on secure software development and the list of attack patterns above.

Another, widely renowned reference work on computer security, is *Computer Security: Principles and Practice* by William Stallings and Lawrie Brown (Stallings & Brown, 2018). While several aspects of secure coding are folded into chapters dealing with conceptual issues not directly reduced to code, but for example database security, containing subsections on SQL injections (Stallings & Brown, 2015, 117-183), it also has a whole chapter on buffer overflows (Stallings & Brown, 2015, 341-378) and on software security (Stallings & Brown, 2015, 379-418), containing technical summaries and analysis of code vulnerabilities.

With reference to the *OWASP Top 10* (OWASP, 2019*b*) and the *CWE/SANS Top 25 Most Dangerous Software Errors* list (MITRE & SANS, 2011), Stallings and Brown emphasize the need for defensive programming skills and efforts to bring this knowledge to developers: "Awareness of these issues is a critical initial step in writing more secure program code." (Stallings & Brown, 2018, 381).

As core definition of what secure coding means they provide the following:

> "Defensive or Secure Programming is the process of designing and implementing software so it continues to function even when under attack. Software written using this process is able to detect erroneous conditions resulting from some attack, and to either continue executing safely, or to fail gracefully. The key rule in defensive programming is to never assume anything, but to check all assumptions and to handle any possible error states." (Stallings & Brown, 2018, 382)

Analytically they categorize issues of secure coding into four sections:

- handling input
- writing safe code
- interaction with other programs and the OS
- handling output

When it comes to input handling, Stallings and Brown start with the issue of input size and buffer overflows, which has its own dedicated chapter in the book. And they importantly highlight that not only direct user input is relevant, but also input by the operating system environment and other programs and routines. While they stress the importance of buffer overflows when it comes to languages like C and C++, little is said about buffer overflows in high level languages like PHP or Python, which are widely used for web application programming (Stallings & Brown, 2018, 384-385). This could be a general blind spot, as most literature and web search results on buffer overflows have to do with C/C++. Even the OWASP page on *Buffer Overflow* currently suggests it might only be an issue related to "C, C++, Fortran, Assembly" (OWASP, 2019*a*) [1].

But while PHP itself is not vulnerable to buffer overflow in the sense that a programmer could allocate a too small buffer and then write too large amounts of input data to it, PHP still uses the underlying OS, usually a web server like Apache, and many other libraries which are often written in C. As just one example we could look at the PHP 5 changelog for version 5.6.40 from 10 Jan 2019. There we find fixes for several buffer overflows in 4 widely used libraries: GD, Mbstring, Phar, and Xmlrpc (php.net, 2019*a*, Section: Version 5.6.40, 10 Jan 2019). In the PHP 7 Changelog we even find a very recent buffer overflow vulnerability in the core (php.net, 2019*b*, Section: Version 7.3.9, 29 Aug 2019). Or another one from 2019 in a core function: "Buffer Overflow via overly long Error Messages" (php.net, 2019*b*, Section: Version 7.3.3, 07 Mar 2019). So it is important that also web developers using PHP are aware of buffer overflow issues and that we should create awareness against a false sense of security in this regard.

What Stallings and Brown write for buffer overflows, is also valid for the other issues regarding input handling: "Writing code that is safe against buffer overflows requires a mindset that regards any input as dangerous and processes it in a manner that does not expose the program to danger" (Stallings & Brown, 2018, 385). They emphasise the importance of correct interpretation of programme inputs, syntactically as well as semantically. They highlight this on the example of injection attacks, including *command injection*, *SQL injection* as well as *code injection*. They also feature cross site scripting (XSS) attacks as a major problem in the area of input handling.

For the category of writing safe program code, they start with the issue of correct algorithm implementation (Stallings & Brown, 2018, 396-398). This is a problem which is harder to evaluate automatically, as the evaluation of correct algorithm implementation most often cannot be specification-agnostic. In the case of the SCRAP prototype, that is developed in course of this thesis, this would mean, that the toolchain would have to know up-front, in some formalized way, what the exercise requirement is, in order to evaluate if the algorithms have been implemented correctly. Fortunately most issues of correct algorithm implementation will be evaluated by existing exercise submission tools, so this is not in the scope of SCRAP. Nevertheless, this could be developed as a potential extension to SCRAP, for cases where (secure) algorithm implementation is the main focus of an exercise. [2]

[1]This is valid for the state of the page on 2019-09-16. The disclaimer of the page says: "This Page has been flagged for review. Please help OWASP and review this Page to FixME. Comment: No real edits since 2009"

[2]Also not in the scope of SCRAP, but noteworthy, as it is an issue "that is largely ignored by most programmers" (Stallings & Brown, 2018, 398), is to ensure that the running machine code correctly corresponds to the algorithmic implementation. This has to do with compiler security and integrity and should also be consideration in information security management implementations, especially in organisations which write and/or compile code themselves. Also in the area of web application security, developers should at least be aware that in the end

Other issues of writing secure program code, mentioned by Stallings and Brown are the *correct interpretation of data values*, the *correct use of memory*, and *preventing race conditions with shared memory* (Stallings & Brown, 2018, 398-400). These can be tested with static code analysis to some extent.

When it comes to interaction with the OS and other programs and libraries, they list the following key issues (Stallings & Brown, 2018, 400-412):

- Environment variables
- Using appropriate, least privileges
- System calls and standard library functions
- Preventing race conditions with shared system resources
- Safe temporary file use
- Interacting with other programs

All of these issues can be addressed from an operations point of perspective as well as from a secure code perspective. In any case developers should be aware, that operators might not handle configuration and execution of their programs in the way they expect them too and that they should - wherever possible - prevent misuse through writing their code accordingly.

For the final category of handling output data, they stress the importance of sanitising the output. They especially highlight XSS as a prominent example where malicious output data can cause substantial harm. The general point to make here is, that one can never fully assume how and what the output data will be used for and therefore always should adhere to a system's specification.

With these examples Stallings and Brown provide a compact guide on what issues to consider for secure coding. Yet, this is a standard reference book introducing students to computer security. When we are looking for a standard reference work that focuses entirely on secure code and provides a systematic analysis of how code can be vulnerable and how to secure it, we have to go back to the early 2000s.

In *Writing Secure Code*, Michael Howard and David LeBlanc lay out all the potential security issues code could have. While all the categories of vulnerabilities listed above - in their terms from the "Public Enemy #1: The Buffer Overrun" to "Internationalization Issues" - are explained in detail, they also include special chapters on issues like socket security and how to implement secure remote procedure calls, how to facilitate the Distributed Component Object Model (DCOM) in a safe way as well as how to protect against DDoS attacks (Howard & Le

they often put unquestioned trust in the web server and its execution environment. But for the case of students learning to write secure web application code in introductory programming courses, we should probably assume a safe execution environment.

Blanc, 2003). While they write from a Microsoft perspective in the early 2000s, and therefore such things as DCOM get extra attention, after conducting my full literature review, this books still seems to be one of the most comprehensive reference books on writing secure code. Of course most of the mentioned tools are now deprecated, but from an analytical perspective it provides a systematic and complete overview on code vulnerabilities and also includes systematic guidance for security testing.

They also include some guidance on threat modelling and present their STRIDE and DREAD models for categorising threats and calculating risks, by decomposing an application and investigating all its subsystems/components. STRIDE is comprised of the following threat categories (Howard & Le Blanc, 2003, pp 83):

- Spoofing identity
- Tampering with data
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

To evaluate the risk of code vulnerabilities the DREAD model is comprised of 5 factors (Howard & Le Blanc, 2003, pp 93):

- Damage potential
- Reproducibility
- Exploitability
- Affected users
- Discoverability

While DREAD is not used by Micrsoft anymore since 2008, there are more modern risk assessment frameworks we could use, like the CVSS or similar scoring systems (FIRST, 2019). Yet STRIDE could be used as a model to explain found code vulnerabilities in a bigger application context.

In their chapter on security testing they transform the STRIDE model to a set of testing techniques and strategies. Beside these, they emphasise the need to “exercise the interfaces by using data mutation” (Howard & Le Blanc, 2003, ch. 19, sec: Building Security Test Plans, subsec: Attacking with Data Mutation). This is done by “perturbing the environment such that the code handling the data that enters an interface behaves in an insecure manner” (ibid). According to their generic test scheme program input data can be tested based on (ibid):

- size, which could be:
  - too long
  - too short
  - of zero length
- content, which might be:
  - random data
  - null
  - zero
  - of the wrong type
  - of the wrong sign
  - out of bounds
  - a mix of valid and invalid data
  - on-the-wire data, which might be:
    - replayed
    - out-of-sync
    - high volume
  - made up fully or partially of special characters, which could be:
    - meta characters
    - script tags and characters
    - escaped characters and code sequences
    - HTML and script code
    - slashes
    - quotes
- the container (e.g. file metadata), which might:
  - be a link and not a regular file
  - not exist
  - exist
  - be restricted in access
  - be not accessible at all
  - have an unexpected name, to wich all the data testing techniques as listed above apply (content and size)

With this rather complete data testing technique and the categories of vulnerabilities listed above, we have a sufficient reference model for testing code for weak security or outright vulnerabilities. Additionally, the next subsection provides an overview of current frameworks and reports on software security and vulnerabilities as they are found in practice.

2.1.1 Vulnerabilities & State of Software Security

Two of the most established and well-known frameworks to categorise existing software vulnerabilities are the regularly updated *Top 10* list by the *Open Web Applications Security Project* (OWASP) (OWASP, 2019*b*) and the *CWE/SANS Top 25 Most Dangerous Software Errors* list (MITRE & SANS, 2011), both also mentioned as main references in (Conklin et al., 2016, Ch. 18, Subsection: Coding Phase).

While the OWASP Top Ten list was published since 2004 in a new version every 3 to 4 years (2004, 2007, 2010, 2013, 2017), the CWE/SANS Top 25 came out in 2009, 2010 and in the last version in 2011. Only in 2019 a new and overhauled version was published as the *CWE Top 25*. The old CWE/SANS versions used "surveys and personal interviews with developers, top security analysts, researchers, and vendors. These responses were normalized based on the prevalence and ranked by the CWSS methodology." (MITRE, 2019). The new 2019 version took a different, more data driven approach "based on real-world vulnerabilities found in the NVD" (ibid).

The *OWASP Top 10* on the other hand is data driven in the way, that real application vulnerabilities are counted, assisted by proper tooling, but the data gathering is based on a more qualitative approach, including an industry ranked survey of vulnerability categories and a public data call to submit vulnerability data (OWASP, 2017, 24).

A main difference between the *OWASP Top 10* and the *CWE(/SANS) Top 25* is the former's focus on web application and risk assessment while the latter uses the full range of application domains and focuses on weaknesses. This is also highlighted in the 2011 version of the *CWE/SANS Top 25*, which includes an appendix on mapping to the 2010 version of the *OWASP Top 10*, stating that "[i]n general, the CWE/SANS 2010 Top 25 covers more weaknesses, including those that rarely appear in web applications, such as buffer overflows." (MITRE & SANS, 2011).

The most recent *OWASP Top 10* list from 2017 includes the following vulnerability classes listed in table 1 (OWASP, 2017).

While XSS vulnerabilities still ranked 3rd in the 2013 Top 10 and went down to 7th position in the recent report, Cross-Site Request Forgery (CSRF) was on position 8 in 2013 and did not make the top ten in the 2017 report. This is because "many frameworks include CSRF defenses, [so] it was found in only 5% of applications" (OWASP, 2017, 4). As introductory programming courses might not facilitate full-fledged web application development frameworks but encourage students to come up with their own solutions and to understand the more basic operations behind complex frameworks we still should not dismiss CSRF as mostly irrelevant.

Table 1: Vulnerability Categories of the 2017 OWASP Top 10.

1.	Injection
2.	Broken Authentication
3.	Sensitive Data Exposure
4.	XML External Entities (XXE)
5.	Broken Access Control
6.	Security Misconfiguration
7.	Cross-Site Scripting (XSS)
8.	Insecure Deserialization
9.	Using Components with Known Vulnerabilities
10.	Insufficient Logging & Monitoring

Similarly, the downfall of XSS is partly due to better detection and tooling, but XSS nevertheless is “the second most prevalent issue in the OWASP Top 10, and is found in around two-thirds of all applications” (OWASP, 2017, 13).

Also in the *CWE Top 25* we still find XSS and CSRF quite prominently beyond the first ten, with XSS even ranking as the second most dangerous vulnerability, as displayed in table 2 (MITRE, 2019).

The report even lists 15 more categories as “Weaknesses On The Cusp” with some of them not completely out of scope for web applications, like “Incorrect Type Conversion or Cast” or “Allocation of Resources Without Limits or Throttling”. But as already the 2011 report emphasized in its comparison to the OWASP Top 10, not all of these categories apply to web applications. Also several of them are more relevant for the configuration and operation of (web) applications, so they are not immediately relevant for learning secure web application development. Of course developers should be aware of those issues. But for the sake of an introductory programming course, these issues might result in information overload.

Similarly, the recent *Veracode State of Software Security* report Vol. 9 lists 20 most common vulnerability categories (Veracode, 2018, 25). Most of the listed categories can also be found in (or as a subset of) those in the *OWASP Top 10* and the *CWE Top 25*. Nevertheless, it makes some of them more explicit, like “CRLF Injection”, “Error Handling”, “Time and State”, and for modern web applications especially relevant: “API Abuse”.

A particularly interesting study in our context is presented by Felix Schuckert, Basel Katt and Hanno Langweg by analysing a broad range of open source codes for code patterns that feature SQL injection vulnerabilities (Schuckert et al., 2017). They use cvedetails.com as a source for vulnerability data. This database was crawled for all CVEs from 2010 to 2016 and checked for GitHub commit links. With the latter the crawler checked for the programming language in use. A general comparison of vulnerability categories and languages found that projects based on

Table 2: 2019 CWE Top 25 Most Dangerous Software Errors.

Rank	Name	Score
1.	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
2.	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
3.	Improper Input Validation	43.61
4.	Information Exposure	32.12
5.	Out-of-bounds Read	26.53
6.	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
7.	Use After Free	17.94
8.	Integer Overflow or Wraparound	17.35
9.	Cross-Site Request Forgery (CSRF)	15.54
10.	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10
11.	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.47
12.	Out-of-bounds Write	11.08
13.	Improper Authentication	10.78
14.	NULL Pointer Dereference	9.74
15.	Incorrect Permission Assignment for Critical Resource	6.33
16.	Unrestricted Upload of File with Dangerous Type	5.50
17.	Improper Restriction of XML External Entity Reference	5.48
18.	Improper Control of Generation of Code ('Code Injection')	5.36
19.	Use of Hard-coded Credentials	5.12
20.	Uncontrolled Resource Consumption	5.04
21.	Missing Release of Resource after Effective Lifetime	5.04
22.	Untrusted Search Path	4.40
23.	Deserialization of Untrusted Data	4.30
24.	Improper Privilege Management	4.23
25.	Improper Certificate Validation	4.06

PHP code yield most results when it comes to SQL injections, compared e.g. to projects based on C/C++ which, unsurprisingly, yield most results for buffer overflow vulnerabilities.

Therefore all vulnerable PHP code samples found on GitHub have been analysed in a manual review process, supported by PhpStorm's data flow analysis tool. Based on this, process pattern categories for SQL injection vulnerabilities were produced. From this analysis a categorisation was created, which describes the found SQL injections by the following origins (ibid, 3-5):

- input sources
- string concatenations
- sinks for database queries
- fixes from the CVEs
- failed sanitisation attempts

Within their sample of 50 CVEs they categorised the sources into the following five functional loci of those web applications:

- HTTP wrapper methods (20 occurrences)
- HTTP methods (16 occurrences)
- custom functions (9 occurrences)
- environment variables (1 occurrence)
- configurations (e.g. files) (1 occurrence)

The concatenations are divided into 44 occurrences of primitive concatenations that only use the core language features and 12 uses of standard functions. The sinks consist mostly of "Self-implemented database connection" (41 occurrences), and 5 unsafe uses of data-represented objects (like DAO or ORM) as well as 3 unsafe uses of standard database drivers.

They also categorised the fixes to the found vulnerabilities into the following five approaches:

- standard methods for sanitisation (21 occurrences)
- use of fixed data types (14 occurrences)
- use of custom sanitisation methods (10 occurrences)
- checking for valid input (e.g. white-/blacklisting) (5 occurrences)
- use of parametrised SQL queries (4 occurrences)

In the area of *failed sanitisation* the majority was due to no use of any data sanitisation at all (32 occurrences), while in 15 cases there was some sanitisation, but incomplete (e.g. not for all parameters). Then they also found three cases, which they put into separate categories: 1) *non-casted variable*, where casting did happen, but the non-casted input was used for the query, 2) *unexpected control flow*, where a check was made, but the consequence was just to set a redirect header, without stopping the script execution, and 3) *comparing different data types*, where a string was checked by comparing it to an integer but using the whole string in the query, including potentially trailing injection code.

In the discussion of their project they highlight how pervasive SQL injections still are and how much this looks like a basic educational problem:

> "The results show different kinds of source code patterns that result in a SQL injection vulnerability. The resulting categories allow to create different source code permutations of SQL injections. Many source code parts from relating CVE reports did not have any sanitization of the inputs. Many developers probably require a better software security education to prevent such issues. For example, the source code from CVE-2014-8351 looked like a classical teaching example of a SQL vulnerability. All relevant parts of the SQL injection (source, concatenation and sink) were not even in ten lines of code. This shows that software security education is indispensable." (ibid, 5)

This features in a very vivid way, that our computer security educational systems have a long way to go, when it comes to secure coding, and it is in line with research on software security education, as presented in section 2.3. It also points to SQL injections as a prime target for analysis, especially in introductory programming contexts, which is reflected in SCRAPs data set for evaluation as presented in chapter 5. In general this study provides a good template for categorising and finding SQL injections. It might make sense to have - in a similar way - a full pattern landscape in relation to e.g. the *OWASP Top 10*. This then could be a basis for a modular open framework for code analysis and feedback.

2.2 Platforms and Tools

This section reviews literature on platforms and tools for static code analysis and how it can be applied (among others) in introductory programming contexts. The following subsection on PHP static code analysis provides an overview of available F/LOSS static code analysers and static application security testing (SAST) tools, which will be evaluated in more depth in the prototype chapter (5).

Focusing on the client side of web applications Anis et al. developed and evaluated a method to ensure the integrity of the JavaScript parts of a web application, that runs in the visitors browser (Anis et al., 2018). Citing the *OWASP Top 10* and the *2011 CWE/SANS Top 25 Most Dangerous Software Errors* reports, they argue that “the attackers focus has shifted from the server-side to the client side” (ibid, 618). Therefore they wanted to “introduce an approach to secure web applications by providing guidelines to the developers to prevent SQL injection, XSS, and resource alteration attacks” (ibid). Their contribution was described as two-fold: First, by developing security policies for web applications, they want to support proper implementation of secure coding. Second, they develop an integrity verification module (IVM) that “prevents code tampering at runtime” by protecting “JavaScript code on the client side [...] from alteration.” (ibid, 619)

As their argument is, that increasing client-side security also helps to increase the overall security of a web application, they focus on issues of SQL injections, XSS attacks and resource alteration. While the former two can and should be mitigated by the backend components altogether, additional mitigation on the client side decreases the overall attack surface of a web application. In case of resource alteration, the focus is specifically on client-side issues. In this case the question is how to prevent runtime alteration of the JavaScript-based client code and also how to prevent injection of malicious code through compromised content delivery networks (CDNs) or other third party resources.

To mitigate against these issues, they propose 5 security policies:

- input sanitisation
- output validation
- least-privilege
- sub-resource integrity
- content security

These policies should be considered by developers throughout the software development life-cycle. But it also should be enforced by their IVM, which has to be integrated in the web application’s code. “It is designed to secure the verification module and to maintain obfuscation during runtime.” (ibid, 621)

The IVM design is based on including several web workers which operate independently from the main UI thread of the web application client and in the background, doing all the hashing and verification, only interacting punctually with the main app. This way a continuous verification of code integrity can be performed without decreasing the responsiveness of the user interface.

For the evaluation of their IVM they used the following four tools: *Vega* for SQLi, header injection and XSS, OWASP’s *Zed Attack Proxy* as an automated scanner and fuzzer as well as for passive scanning and forced browsing, *Skipfish* for security threats and vulnerability reports, and *JBroFuzz* for automated fuzzing.

They used the combination of these on 22 different web applications, each from a different application area (e.g. blogging, booking system, chat, e-learning, data analysis, restaurant management, Pokemon Go, etc.).

Comparing the prevention rate of all the web apps without the IVM to a version of them that includes the IVM, they found that the IVM increases the attack prevention rate significantly (24% for SQL injections, 31% for XSS attacks, and 43% for resource alteration attacks).

Through the use of more web workers in parallel, the optimum in their experiment being 16, the time for integrity verification can be brought down to 0.04 seconds and attack reporting happens between 2.5 and 4.5 seconds after an attack is initiated.

Overall this work seems quite promising, but is not in scope for SCRAP. Nevertheless it shows that secure coding should also be fostered for frontend components, especially as this area is often neglected in security considerations. Of course the backend components have to mitigate against all those issues, but that does not mean that secure coding can be ignored as soon as it comes to JavaScript-based client components.

A graph-based approach to static analysis is evaluated by Sahu and Tomar. They created a "multi-relational graph-based interactive system" that could be used to do code inspection while developing (Sahu & Tomar, 2017). In their work they list a lot of other work on code analysis and vulnerability detection, not only focusing on PHP specifically. But the most notable mentions of SAST tools are *DevBug*, *RATS* and *RIPS*, which they also use as a benchmark to compare their own tool against.

To develop rules of secure coding standards that can be used in their automatic graph-based analysis, they draw on the OWASP Development Guide and Howard and LeBlanc's *Writing Secure Code* [3] (ibid, 887).

In their rationale the majority of web applications are vulnerable either due to "improper handling of language-specific functionalities" or "poor style of writing code". With the latter they mostly mean "code by the less experienced programmer[s]" who would be encouraged by "[f]ree available templates, open-source web development platforms, minimal hosting cost and their ad hoc nature" (ibid) to write more and more web applications with a primary interest "to build a user-friendly interface rather than creating the secure system" (ibid, 888). [4]

Sahu and Tomar identified the following four PHP features with potential threats (ibid):

- "Type safety/ type juggling"
- The php.ini file
- "Dynamic code inclusion"

[3]Analogous to my own assessment, they attest this reference to be an "excellent" one when it comes to secure coding.

[4] I would argue that poor style of writing code is not primarily the problem of inexperienced or beginning programmers, but an endemic problem of an industry focusing on quick releases and decreasing development costs. Of course new programmers might find it even harder to address secure coding issues, use well-designed code patterns, and come up with creative and nevertheless secure solutions for tricky problems. But as my own literature review on secure coding education suggests, this problem is pervasive throughout all educational stages and industries.

- “Dynamic command execution”

While they describe their approach in a way that makes clear how they approach the problem and what the principle for the design of their tool is, they are not at all stringent and conclusive on how the tool works in detail and how it identifies vulnerabilities. Also the sample code and GraphML snippets seem to not be syntactically correct, as if they made some copy-paste-errors. This makes the approach unpracticable, if not unusable for integration into other work, nevertheless it seems like a novel approach worth further investigation.

For the comparison of their tool to RATS, RIPS and DevBug on the basis of an unlabeled data set they used the *Damn Vulnerable Web Application* (Dewhurst Security, 2020) and the *OWASP Mutillidae Project* (OWASP, 2020*c*). But in describing their results they do not describe exactly how they extracted code snippets from their labeled data set (CVE, NVD, Syhunt Vulnerable PHP Code, OWASP Filter Evasion Cheat Sheet) and how they actually compare it, except than mentioning that for “cognitive performance analysis, remaining performance metric of developed interactive system and three existing scanning techniques are computed from assembled labeled dataset, and results are compared” (Sahu & Tomar, 2017, 892), without any further reference on the applied methodology.

According to their analysis their tool performs quite well (ibid, 894). However, their results are not reproducible based on their descriptions. Nevertheless the approach seems interesting. Due to absent methodological explications and source code, this tool and approach cannot be further used for SCRAP.

A particularly interesting platform for secure coding integration in software development is presented by Heymann and Miller in a Tutorial at the 2018 IEEE Secure Development Conference. The tutorial targets “developers wishing to minimize the security flaws in the software that they develop” and starts with “common vulnerabilities found in middleware and services [and d]escriptions of each type of vulnerability” (Heymann & Miller, 2018, 124). The second part focuses on automated assessment tools and how to work with them.

In their final section they present the *Software Assurance Marketplace* (SWAMP), which is an initiative as well as a platform trying to increase secure coding practices by providing on-the-go and integrable code analysis capabilities. They host https://www.mir-swamp.org, a free to use analysis platform, and the *SWAMP-in-a-Box*, a stand-alone application to run on ones own infrastructure.

SWAMP incorporates a variety of F/LOSS tools and also a few commercial tools to do code analysis. While the majority of them focus on Java, C/C++ and .NET, Ruby and Python also are well supported. Additionally under the rubric of *Web Scripting* they deploy several linters for HTML, CSS, XML and Javascript, as well as static analysis tools for Javascript and PHP. For the latter they use *PHP_CodeSniffer* and *PHPMD* (Morgridge Institute for Research, 2020).

2.2.1 PHP static code analysis

To gain a sufficient overview of tools and platforms for static (security) analysis of PHP code, I gathered several collections from web searches, the reviewed literature and through references, as presented in table 3. The first column of the table includes a tag, which is used to refer to in the table on specific tools and platforms [5].

Table 3: Reference lists for static code analysis tools.

Tag	Description	Reference	Notes
AWESOME	Awesome Static Analysis repository	(Endler, 2020)	
CERN	CERN static code analysis tools	(CERN Computer Security Team, 2020)	refers to SWAMP
OWASP	OWASP source code analysis tools	(OWASP, 2020*e*)	refers to AWESOME as more comprehensive
SAMATE	NIST source code security analyzers list	(National Institute of Standards and Technology, 2020)	refers to AWESOME and WPLIST
SWAMP	Software Assurance Marketplace list of tools	(Morgridge Institute for Research, 2020)	
WPLIST	Wikipedia list of tools for static code analysis	(Wikipedia (EN), 2020)	

In their entirety, those lists provide a vast and diverse overview on static source code analysis tools and platforms, covering a lot of different languages. For the purpose of an evaluation for potential use in SCRAP, I screened all lists for PHP related analysers with a F/LOSS license and gathered information on its development status and whether it provides a documented API. The result can be found in table 4, which also lists if those tools specifically aim at PHP code or if they cover several languages. [6]

While the focus was an F/LOSS tools, *Coverity* and *RIPS* have been included in this overview, as RIPS is a direct successor of its 0.55 F/LOSS version and Coverity is included due to being widely used for free in F/LOSS contexts through its GitHub integration and via Coverity Scan. RIPS was originally specific to PHP and this focus lives on in the commercial version, although the latter also supports analysing Java code.

[5]For the instantly curious readers, it also features as a direct link to the resource on the web.

[6] The threshold for a yes in the column *API* is if there was a documented way to use the program not only as a stand-alone application but (ideally) through a RESTful web interface. The threshold for a yes in the *maintained* column is a release within the last 6 months (seen from Februar 2020). If the tool is not maintained according to this criterium, the year of the last release is provided after the version number.

Table 4: Overview of F/LOSS static PHP analysers.

Tool	API	PHP-specific	Version	main-tained	F/LOSS
Coverity	yes	no	-	-	no
graudit	no	no	2.3	yes	yes
Parse	no	yes	0.8 (2018)	no	yes
phan	no	yes	2.5.0	yes	yes
PMF (PHP Malware Finder)	no	yes	0.3.5	yes	yes
phpcs-security-audit v2	no	yes	2.0.1	yes	yes
PHPSA	no	yes	0.6.2 (2016)	no	yes
PHPMD	no	yes	2.8.1	yes	yes
PHPStan	no	yes	0.12.11	yes	yes
PHP_CodeSniffer	no	yes	3.5.4	yes	yes
php-sat	no	yes	alpha (2006)	no	yes
Php Inspections (EA Extended)	no	yes	4.0.3	yes	yes
Progpilot	no	yes	0.6.0 (2019)	no	yes
RATS	no	no	2.4 (2013)	no	yes
RIPS 0.5	no	yes	0.55 (2015)	no	yes
RIPS	yes	yes	-	-	no
SonarQube	yes	no	8.1	yes	yes
WAP	no	yes	2.1 (2015)	no	yes
Yasca	no	no	2.2 (2010)	no	yes

graudit and *PMF* both are more of a kind of pseudo static analyser, as they only work with pattern matching. But PMF is explicitly focusing on PHP malware and dodgy code, so it could be useful, also due to its use of Yara rules. This tool was not found on any of the mentioned lists but through an auxiliary web search when working through the lists and the different tools.

php-sat was a Google Summer of Code project, which was apparently never developed further. But the description of its genesis is quite notable:

> “The first source of inspiration came from my work as a assistant at the course ‘internet programmeren’ (Internet Programming) (2005,2006) at my University department. I noticed that a lot of students where not aware of the security problems involved when programming PHP for the web.” (Bouwers, 2006)

Php Inspections (EA Extended) is a general analyser in form of an IDE plugin for PhpStorm/Idea. Therefore it is not tested in this study, but it could be used as a reference benchmark in future and related research.

Among all other analysers that are still maintained, only *SonarQube* has a specific focus on security. And it also is the only F/LOSS solution that provides an API exposed through a web service. This makes it the primary candidate for further evaluation for the SCRAP prototype. Nevertheless *graudit*, *phan*, *PMF*, *PHPMD*, and *PHPStan* are interesting candidates for comparison regarding the hit rate and false positives for found code vulnerabilities. *PHP_CodeSniffer* is also general analyser but with *phpcs-security-audit v2* there is a security specific rule set for it, which makes it additionally interesting.

The concrete comparison and analysis of these tools, tied to its potential integration in SCRAP, is described in the prototype chapter 5. The full analysis and its accompanying data can also be accessed in a separate git repository on GitLab and is linked to on the SCRAP project page: https://scrap.tantemalkah.at.

2.3 Software Security Education

When it comes to fixing the problems of our insecure cyber landscapes, a lot of money, time and energy is invested into building better tools and toolchains to counter attacks and improve software development. But while it is important to fix vulnerabilities from an early start on and to educate users and to set organizational measures, it might be as important and maybe even more efficient to invest in developers’ education on secure coding. In this section I explore current research on software security education, which suggest that an early investment to integrate secure coding principles and defensive programming into programmers education is one of the major factors where we could improve cybersecurity substantially.

At the 2013 IEEE Global Engineering Education Conference and the 2013 3rd Interdisciplinary Engineering Design Education Conference, Alexander Uskov reports on design, implementation and evaluation of the “state-of-the-art undergraduate and graduate curriculum and courseware” in software and web application security (SWAS) at Bradley University (Uskov, 2013*b*) (Uskov, 2013*a*).

Uskov describes the contents and evaluation results of a course called *Software and Web Application Security (SWAS)*, which they proposed and developed from 2009 to 2010 and taught from 2010 to 2012, up to the evaluation 2013. The evaluation highlights common attack patterns at the time and refers to the 2008 and 2013 *ACM/IEEE Computer Science Curriculum: Information Assurance and Security* requirements. To meet the needs for increased skills in secure coding, the “Department of Computer Science and Information Systems (CS&IS Department) at Bradley University (Peoria, IL, U.S.A.) created undergraduate and graduate academic programs (concentrations) in software, Web, and computer network security” (Uskov, 2013*b*, 2). One such course in these programs is the SWAS course, which is detailed and evaluated in the paper by Uskov. In it they stress the need for practical hands-on teaching with an appropriately developed course framework in order to “provide students with deep knowledge and excellent technical hands-on skills for each type of computer attack discussed in a class, and b) prepare students to deal with advanced computer attacks (i.e. Attacks 2.0) in real-world environment.” (ibid, 3)

Uskov’s work shows that we have to integrate secure coding concerns early on in introductory programming courses - as highlighted below by other research. But it also shows that additionally we need explicit courses on software and web application security. The question for computer science departments then is, if these extended courses should be mandatory. The secure coding research review by Uskov certainly mandates this, as *security as add-on* never works in a satisfactory way.

While not directly on issues of secure coding, Juha Sorva describes struggles of students in introductory programming courses (Sorva, 2013). Sorva provides a good overview and insights which might be very much valuable for adopting secure coding practices in introductory programming contexts as well. Therefore I want to describe Sorva’s research in a bit more detail.

The article focuses on the use of *notional machines* in programming education. But it also includes an extensive literature review on introductory programming education, as it springs from Sorva’s dissertation (Sorva, 2012). Both aspects are interesting for SCRAP and the context of integrating secure coding into early stages of computer science curricula.

Notional machines are abstract models of computers that execute specific programming language code. While a notional machine could be a quite generic abstraction of a common personal computing system, usually notional machines merge abstractions of the computer hardware, operating system and the specific programming language in use. Originally they were introduced to computing education research in the 1980ies by Benedict du Boulay as idealized computers “whose properties are implied by the constructs in the programming language employed” (Boulay, 1986, cited by) (Sorva, 2013, 2). An example illustrating the abstract and often language-specific character of notional machines is provided by Bruce-Lockhart and Norvell with the code in listing 1 (Bruce-Lockhart & Norvell, 2007, cited by) (Sorva, 2013, 2):

```
int x=5;
int y = 12;
int z;
z = y/5 + 3.1;
```

Code 1: Example code segment by Bruce–Lockhart and Norvell to explain notional machines

While we find four instructions in terms of the programming language, the actual machine will do a quite different number of instructions. And while the first three lines instruct the compiler, the fourth line holds a lot more instructions for the actual computer (finally the CPU's ALU), as there are high-level concepts applied in this seemingly simple algebraic statement, such as automatic type conversion and truncation.

In this way the model of a notional machine can be heavily influenced by the features of a programming language and programming paradigms in a broader sense. Also different notional machines can be created for a single programming language, providing low or high levels of abstraction. In any way those notional machines are devices that can be used to facilitate a students understanding of how programmes are executed. This is also important for secure coding, as "[i]ncorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs". (Sorva, 2013, 4) This combines with an educational landscape in which "introductory programming courses are not particularly successful in teaching students about fundamental concepts" (ibid). The literature reviewed by Sorva "suggests that many of the problems of novice programmers are related to inadequate understandings of the notional machine, and especially to the "hidden" processes that are not directly apparent from program code" (ibid, 7)

The extensive literature review by Juha Sorva on notional machines and what mental model theory, constructivism, phenomenography, and threshold concept theory as well as empirical research in these fields suggest in regard to learning to programme, points towards a need to rework introductory programming courses. This confirms my own experiences - the older ones about being a participant in introductory programming courses and workshops as well as the newer ones of teaching and facilitating introductory programming workshops. It also confirms anecdotal evidence about experiences in learning to programme by friends, colleagues and participants in introductory programming workshops.

One of the main problems seem to be limited resources, primarily of time. But Juha Sorva suggests to address and use notional machines (more) explicitly and to focus on key concepts and obstacles, because a time-wise investment in this area will improve learning and reduce failure rates in attempts to create code that solves specific problems and to learn new coding paradigms and patterns.

But we should not forget that time economies of teachers in introductory programming courses might differ greatly from those of their students. In a vague sense the review suggests that the use of notional machines will also reduce the time needed for teachers to explain specific aspects and tune them to those aspects which yield most gain for students' knowledge. Nevertheless in order to achieve significant change in how notional machines are used in introductory programming at least in the beginning more time investment will be needed by teachers and the institutions they are teaching in.

All of that points towards a similar problem we face with secure coding education. While everyone accepts that it should be introduced early on, such as the concept of notional machines, little is done to change the structural context which is needed to do so. The structural context in this case is the design of the computer science curricula, the amount of time given to teachers and students to learn and the organisational policies and resources devoted to create integrated and holistic learning approaches and spaces.

Notional machines improve the understanding of what specific programming instructions and code patterns do in the final execution environment. They make a student's understanding of a specific programming language and paradigm more holistic. In the same sense they could improve a student's understanding of security concepts and awareness for and understanding of vulnerabilities of specific instructions, functions and code patterns.

Out of Juha Sorva's work arise several pedagogical suggestions, of which the following are particularly interesting not only for introductory programming education as such, but also for including secure coding skills and awareness:

- *Use conceptual models and make them explicit*: In every introductory course on programming, there will be a notional machine anyway. It is "implicit in the programming language used" (ibid, 20), and students will come up with some model themself, however appropriate. So using and making explicit a notional machine will mitigate a lot of misunderstanding and hurdles in learning to code (securely). One way to do so is to use conceptual models, which means that "instruction should start with the underlying model [of a programming language] before proceeding to the abstractions founded on it" in order to avoid the "ill-fated construction of intuitive knowledge about the computer and programming language semantics" (ibid, 21).

- *Inform students about threshold concepts*: There are several concepts involved in learning to programme, which are hard to grasp at first encounter and which, to be fully understood, need a certain threshold of experience and tacit knowledge in how to deal with uncertainty and the frustrations that sometimes arise from coding without having yet grasped a threshold concept. Therefore "teachers should seek to inform students about the existence of threshold concepts and liminality, increase students' metacognition about their liminal states" (ibid). In other words, students "should be helped to become aware of the ways in which they presently think and practice, and motivated to transform those ways" (ibid).

- *Visualize*: To explain the behaviour of a notional machine visually improves understanding significantly. This might either be done by using drawings, flow charts, diagrams, etc.

on black/whiteboard and with presentation software, or it even might be accomplished by using visual debuggers “a step-by-step trace of an input program as it is executed, making explicit the flow of control, the values of variables, frames on the call stack, and so on” (ibid, 22). There are even specific educational debuggers for that (Sorva et al., 2013).

- *Foster active learning*: While one can understand systems to a reasonable degree by monitoring or observing them, advanced understanding and creative problem solving skills arise from controlling or manipulating systems. The studies Sorva cites highlighted something already in the 1980ies and 1990ies, for the particular case of monitoring/-controlling complex systems, that seems quite similar to what Hooshangi et al found for attack vs. defense code (Hooshangi et al., 2015). While the former demands the student to understand how a system works to spot either malfunctions or vulnerabilities, the latter demands the student to interact with the system in a way that increases the understanding of its complexity as well as the awareness for malfunctions or vulnerabilities. So whenever (secure) coding is taught, there should be an emphasis on students hands-on experiences.

These are all important points to be considered when implementing and integrating secure coding courses and modules.

More specifically than focusing on programming education in general, software security education was one of the main topics of the *9th World Conference on Information Security Education* (WISE9), which took place in Hamburg in 2015, in conjunction with the *IFIP International Information Security and Privacy Conference* and was organized by the *IFIP Working Group 11.8 – Information Security Education*. The proceedings contain three papers whose main focus is the process of educating new programmers in university contexts (Bishop et al., 2015).

Aside from the concrete process of implementing some feature by programming code, security can already be a topic before people start to code. This is highlighted by Johan van Niekerk and Lynn Futcher, as they analyse software design patterns in terms of security considerations:

> “Software design patterns are often used to address a commonly occurring problem through a ‘generic’ approach towards this problem. The design pattern provides a conceptual model of a best-practices solution, which in turn is used by developers to create a concrete implementation for their specific problem.” (van Niekerk & Futcher, 2015, 75)

But while these patterns would be a perfect starting point to integrate security from the start on, most design patterns found in use “do not include security principles as part of the generic solution towards the commonly occurring problem.” (ibid, 75-76).

Therefore Niekerk and Futcher propose one example of an improved software design pattern that includes security aspects. To demonstrate their approach they choose the Model-View-Controller (MVC) pattern, by adding a verification and validation box between the view and the controller, encoding and integrity filters between the view and the model, and timeouts, logging and a AAA & trust a box between the controller and the model.

They also stress the point that security should never be added "as an afterthought" (ibid, 76), but be integrated from the start and in all stages of software development. For this they also refer to *Writing Secure Code* (Howard & Le Blanc, 2003), who argue that implementing security as an add-on is usually much more expensive and even might break the original application logic and user experience.

By using their improved and "security conscious" model of the MVC design pattern, the topic of security will be integrated "during every discussion regarding an n-tier design" and by "including security considerations as a default into every web application (or similar) development project, students will get substantial exposure to the issues that should be considered" (van Niekerk & Futcher, 2015, 81). From a teacher perspective the integration "of security principles into existing design patterns can be a powerful tool [...] to improve the teaching of secure software design" (ibid, 82).

While the issue of integrating security already at design-time has its own educational value, as important is the issue of providing feedback to students throughout the whole process of learning to programme. And this feedback should be designed in a way, that helps students to adopt secure coding habits. This was the focus of Raina, Tayler and Kaza, who provided insights into how they improved the materials for coding coursework within the CS0 level courses at Towson University (Raina et al., 2015). They identified *skimming* and *skipping* of the materials by students as the core issues with the old 1.0 coursework modules. For their improved 2.0 modules they used *segmentation* to reduce skimming and skipping effects: "instead of presenting large amounts of hypertext content at once, the content should be broken into smaller chunks and presented one idea at a time on a single screen" (Raina et al., 2015, 66).

Another focus of their project was also to increase interactivity by using *dialoguing* and *controlling* with immediate feedback to the learner. With reference to (van der Kleij et al., 2012) and (Thalheimer, 2008) they classify feedback on the amount of detail it contains into three types (Raina et al., 2015, 66):

- knowledge of results (KR): only the information if an answer is correct or not
- knowledge of correct response (KCR): as KR, but also includes correct answer
- elaborate feedback (EF): as KCR, but also includes explanation

Controlling means to allow the learner to set the pace of the learning and presentation process.

Applied to SCRAP, the type of the feedback is already set to be immediate and allows for controlling, as the feedback happens, when the student submits an example. Regarding the detail, we should aim for an elaborate feedback, but be aware how to present it, in order to avoid skimming or skipping. Therefore feedback should be summarized with further information available by links to the platform.

In general the results of (Raina et al., 2015) - in line with the whole literature review - recommend to not implement the secure coding part of programming education as an add-on but to integrate it from the start. For their project they used Stanford University's class2go web-based application, which is a Django-based open source framework (Stanford Report, 2012).

This remark about the perils of security as an add-on is an important one, as the SCRAP toolchain could also just be used merely as an add-on to existing coursework. Nevertheless the prototype toolchain developed in this thesis can be used in secure coding integrated coursework as a plug-in or add-in for existing exercise submission systems. It just has to be stressed that most value will be added to existing coursework if it is reworked in terms of its didactic design and methodology.

Another study by Raina et al. (2016) extends the study mentioned before (Raina et al., 2015) by examining how student learning is improved and concept retention is increased through the use of segmented and interactive learning modules (Raina et al., 2016).

Although I think the sample size in their study is rather small and therefore the results should be treated tentatively, their study is in line with other research and points out that in terms of learning retention there is no significant difference between more classical linear learning materials and their improved segmented and modularized materials. But in terms of the application of learned knowledge, the improved materials achieved much higher score due to their higher interactivity and sequences where students had to actively apply the knowledge before moving on to succeeding sections.

While they did not research the effects of the modularization and segmentation on skimming and skipping effects in this study, they plan on conducting further research to study these effects in comparison of their old an new modules.

A broader perspective on the state of software security education in the IT educational system is provided by Jøsang, Ødegaard and Oftedal (Jøsang et al., 2015). Their argumentation starts with the observation of a general industry consensus that security has to be integrated into the software development life cycle. But while increasing efforts are put into technical and organizational measures to increase cybersecurity little is done to increase the secure coding awareness of developers who received technical college or university education in IT. To emphasise the importance on putting more efforts into integrating secure coding principles in IT education, they use a probably bold but ravishing analogy:

> "As an analogy, it would of course be irresponsible and even unthinkable to educate building architects and civilly engineers without giving them adequate knowledge about fire safety, otherwise the buildings in which we work and live would be full of firetraps. Likewise it is irresponsible to offer IT programs at universities without compulsory modules in information security. Unfortunately, still today many IT grad-

> uates leave university and go into industry without any competence in information security. Despite their great skills in programming and IT design, without skills in security these IT graduates will necessarily build vulnerable IT solutions." (ibid, 54)

To highlight the flaw in the IT educational system they compare different software development models, with a focus on the waterfall model and the agile model as two ends on a spectrum between large-scale/heavy-weight and flexible/light-weight models.

While security aware versions of these models are already established, all these models rely on developers being aware and knowledgable about secure coding issues. As an example, the *Microsoft Security Development Lifecycle* - representing a secure waterfall model - explicitly puts training and the need for developers to be educated about secure code before the main project start. Agile software development models on the other side just implicitly assume that developers are educated in secure coding. (ibid, 61)

The task of secure software development frameworks is to reduce vulnerabilities. But while it is "of course impossible to completely avoid generating security vulnerabilities during system and software design [...], the state of cybersecurity can be significantly improved by reducing both the number and the severity of security vulnerabilities generated." (ibid, 56)

This is the main argument for demanding the integration of secure coding principles in all programming education and training contexts, especially in technical colleges and universities. Therefore they also conclude with a call to put more efforts into securing coding education: "If a university offers an IT education program with insufficient security, then that university is part of the problem of causing cybersecurity vulnerabilities. It is time for all IT education institutes to become part of the solution." (ibid, 62)

For maximal effectiveness of such an integration it should happen as early as possible in every educational track on software engineering. As Chi et al. argue, "the earlier students learn secure coding concepts, even at the same time as they first learn to write code, the better they will continue using secure coding practices" (Chi et al., 2013). As they put it, we should even broaden the issue to the whole field of STEM curricula, "not to make every STEM student a security expert, but to make them aware of common vulnerabilities and ways to avoid them." They also stress that "[w]riting secure code is an essential part of secure software development" and that "vulnerabilities discovered later in the development cycle are more expensive to fix than those discovered early." (ibid, 42)

A major problem in their view is that the student's main goal in programming courses is to quickly come up with code that works, while not paying too much attention to error handling, failure modes and the validity of user inputs. In order to address this issue they built teaching modules that are fit to be integrated into programming courses for STEM students. These modules draw on static code analysis tools to evaluate students' code, which they categorise into the following four core techniques applied for the analysis (ibid 43):

- Pattern matching

- Semantic analysis
- Symbolic execution
- Abstract interpretation

This categorization also creates a spectrum of increasing sophistication of those tools and techniques from quite simple pattern matching methods to advanced techniques of abstract interpretation of code.

As their modules build on making the students use static analysis tools, they reinforce “the concept of the vulnerability and practice skill at using the tool on a specific source code” (ibid, 44) When we take this into account for SCRAP, we have to avoid creating a situation where students start to just rely on pushing their code to some destination and then receiving some feedback if the code is fine or not. So the generated feedback should be transparent in a way that it includes information on how the code was scanned and analysed, and how they could do so on their own.

On top of 4 of their web based modules for STEM students to learn to apply static code analysis and write secure code, Chi et al. conducted a survey with 70 STEM students as participants, 40% studying computer and information science. Their main result shows that hands-on experience significantly improves student awareness of and self-perceived familiarity with secure coding (ibid, 45-47).

A different angle on learning to code securely is taken by Hooshangi Et Al in an analysis of attack and defense code written by 75 students at New York University, as an assignment in an *Introduction to Security* class. For their assignments, students were tasked with writing a “defense monitor to stop a user from reading data, writing over existing data, or writing new data to the end of a file if there was no permission to do so” (Hooshangi et al., 2015, 2). After they completed this first assignment, they were tasked with writing attack programs (one or many) to test and circumvent the defense monitors submitted by others in the first assignment. The class, in which this assignments took place was a “senior/first-year graduate level class designed for Computer Science BS / MS students and also for Cybersecurity MS students” (ibid, 3).

Their most important finding is that “students who learn to write good defensive programs can write effective attack programs, but the converse is not true” and that “greater pedagogical emphasis on defensive security may benefit students more than one that emphasizes offense.” (ibid, 1). They analysed how unique students’ solutions were and if attack and defense abilities correlated. Regarding the uniqueness they found that about 80% of the student submissions are unique, which “shows that secure coding is a complex problem and different students find different solutions.” (ibid, 4) That those students who wrote good defense programs also wrote successful attack programs but not vice versa, they explain in the following way: “Having the ability to defend entails being able to consider, and handle, possible attacks. However, attack ability is not indicative of talent with defense.” (ibid, 5) While their “experiments do not prove causation” (ibid, 6), their research ties in with those works that argue for including secure coding early on in introductory programming courses, as understanding vulnerabilities increases the complexities students can bring in to finding creative coding solutions.

This is also strengthened by Teto et al., who focus on I/O based vulnerabilities and argue for the integration of *Defensive Programming* (DP) into programming and general computer science education (Teto et al., 2017). They demonstrate the mitigation strategies of DP in the case of Cross-Site-Scripting attacks, by employing rigid input validation, specifically by using filtering, regular expression matching, whitelisting and blacklisting as validation tools. Regarding output sanitisation they highlight escaping and encoding as topics that should be included into a programmers mindset.

In their closing remarks they argue, that we have to "foster the Defensive Programming mind-set in CS and related degree programs by both training future software developers in cyber-security awareness and equipping them with fundamental tools to fight cybersecurity attacks" (ibid, 8). As a pragmatic way to do so, they suggest to integrate the OWASP ASIDE/ESIDE project into educational coding environments. While the ASIDE part is directed more towards professional software development, the ESIDE branch "focuses on help educating students secure programming knowledge and practices" (OWASP, 2016)

The mentioned ASIDE project was developed and evaluated for teaching secure coding by Jun Zhu, Heather Richter Lipford and Bill Chu at the University of North Carolina (Zhu et al., 2013). Their starting point was the realisation that "[d]espite the importance of secure software development, there is little research in how to effectively incorporate the training into existing degree programs and computing curricula." (ibid, 687) While "security education researchers are recognizing that security education must be threaded throughout the entire computing curriculum" the main obstacle to accomplish this is, that "CS faculty have never been trained in secure programming" and "existing courses may not have the time or resources to cover additional secure programming topics on top of already extensive course content." (ibid)

To tackle the problem Zhu et al developed a prototype for an Eclipse plug-in, called "Assured Software IDE", or in short *ASIDE*. The idea is that students will be coding for their programming assignments with an IDE anyway, so they could use Eclipse and the ASIDE plug-in. While they only work on their original assignment, they get additional on-the-fly feedback regarding secure coding in relation to their own code.

In terms of their research ASIDE is a "proof-of-concept Eclipse plugin for Java", which is evaluated in context of an "advanced Web application development class". It works as a long-running background process, scanning all project code files for "patterns that match pre-defined heuristic rules of security vulnerabilities". When ASIDE finds a vulnerable code segment, it marks the corresponding line(s) with a warning icon and also highlights the code in question in red. It then provides several options to choose from. Besides ignoring the warning, the programmer can also let ASIDE generate a secure template for the code in question. There is also compact information in form of explanation pages within Eclipse and links to additional sources.

As ASIDE was initially created for professional developers, one focus of this study was to evaluate the specific needs of programming students, in contrast to professional developers. While the latter primarily made use of the automatic code generation feature, students have been less willing to do so and wanted to have more explanations and examples. Therefore the study at hand focused on how students used the read more links and the provided explanations and how and if they learned secure programming through the use of ASIDE.

In their evaluation they looked at how students adopted to the warnings and explanations provided by ASIDE, which worked in the following way: "For input validation and output encoding, ASIDE provides automatic code generation fixes, but for dynamic SQL statements, ASIDE only provides warnings and explanations. Students must then manually modify the dynamic SQL statement to a prepared SQL statement." (ibid, 691) They found that 60% of the students adopted their code by using prepared statements, after they encountered these warnings about SQL injections and reading the explanation pages. And while "the course lectures, examples, and textbook only provided dynamic SQL statement examples [...] even a short exposure to ASIDE did result in a successful change in practice beyond merely automatically generating code." (ibid)

One of the important findings also was, that students liked the explanation web pages, as "they were easy to understand and very helpful" (ibid) in grasping the concepts and secure mitigations against code vulnerabilities . This highlights the importance to invest in good explanations and reference materials.

It seems that integrating a tool like ASIDE in the course environment is a good approach to compromise between not just doing security as an post-hoc add-on and also not taking away resources from established courses. For SCRAP and any other tool that tries to generate awareness after code was written and submitted, it is just the more important to have high quality explanations and materials, otherwise the adoption rate might be quite low.

A follow up study in 2015 was conducted by Michael Whitney and the three authors of the former study from 2013. While in 2013 the tool was still called *ASIDE*, short for *Assured Software IDE* and reflecting its origin as a support tool for professional developers, now it was called *ESIDE*, short for *Educational Support in the IDE*. The focus of the tool shifted towards providing "instructional guidance and educational materials about secure coding in a contextually based real-time manner as students are writing code" (Whitney et al., 2015, 60).

While the first study only evaluated the use of *ASIDE* in a 3-hour lab study, this study follows up with an evaluation of *ESIDE* in two field studies within the same advanced web development course, one covering the period of a single course assignment, the other covering the whole semester. The results of these studies "demonstrate that ESIDE does raise security awareness, but that the timing of the tool's introduction, and the support of instructors for tool use may be critical for motivating students to learn and practice secure coding skills" (ibid) Notably they point out that in 2015, still, there "has been limited research in incorporating security principles across a standard computing curriculum” (ibid, 61)

They also list a variety of approaches to feature secure coding in computing curricula (ibid, cf. 60-61):

- adding lectures to existing (programming) courses
- adding elective courses on secure coding
- security tracks in the curriculum
- whole programmes in information security

But a major problem that persists with all these approaches is, that it caters to the interests of those who explicitly want to include security into their work and leaves out the majority of those who learn programming within their curricula and come to an understanding that security is something they could add on later, if they develop a special interest.

ESIDE should tackle this problem by integrating secure coding awareness and solutions within the assignments, while the course content itself would not have to be redesigned in a major way and also the course instructors did not need to develop explicit secure coding expertise.

While the results showed that the use of ESIDE increased students’ awareness, the submitted code for the assignments did not necessarily improve in terms of security. The main factor for that was time constraints and that students often would have liked to follow up on ESIDEs suggestions and hints, but have been primarily struggling with submitting a functional solution until the assignment deadline.

So far we have seen that good learning materials and explanations on how vulnerabilities are introduced in code and how to mitigate them are key to successful secure coding education. This will be crucial for SCRAP as well. Nevertheless not only (online) materials are important but also the course design and how interactivity is fostered online or in person.

A study comparing an on-campus course and a MOOC course on software security, using the same contents, is provided by Theisen et al. from North Carolina State University (Theisen et al., 2016). Their aim was to find out how both versions (in-person and MOOC) draw in different sets of students and how effective the learning is in both types of courses. Another goal was “to assist educators in constructing software security coursework by providing a comparison of classroom courses and MOOCs” (ibid, 1).

Drawing on educational research they suggest to use a *flipped classroom* setting for the on-campus course, where the classic lecture parts are transferred to an online space, through videos or podcasts, potentially extended with text, in order to use the physical presence in the classroom so that “students interact with the instructor and each other during class time” (ibid, 2). This increases student engagement. And as this is a blended learning approach, a lot of work that is put into material preparation can be reused and also implemented in MOOCs and other forms of online learning.

One suggestion from research into MOOC structuring for efficient learning is that “lectures incorporated into MOOCs should be broken into manageable chunks for students, typically 5-15 minutes each” because this is “more manageable for students and prevents their minds from wandering as often.” (ibid). In either case (on-campus vs. MOOC), fostering discussion among students helps to increase engagement. This should be actively facilitated by lecturers using open-ended questions. Theisen et al. specifically refer to the *Coursera* guide which “suggests seeding questions during lectures” and that a “few minutes of lecturing should be interrupted by questions periodically.” (ibid, 3). They also drew on peer reviews by students, as they “provide more in-depth projects and assignments for students without overwhelming the course staff, as non-automated assignments are unfeasible once courses hit large numbers of enrolled students” (ibid).

The focus of the offered *Software Security* course lay on types of vulnerabilities and how to prevent or remove them. The courses aimed at increasing awareness and skills in *security risk management*, *security testing*, *security requirements, validation, and verification*, as well as in *secure coding techniques*. All the materials from the on-campus course are available online (Williams, Laurie, 2016). As a plattform for the MOOC they chose *Google Course Builder*, which is an online education platform developed and provided by Google under an open source license (Apache 2.0). They chose it due to previous courses at North Carolina State University having been deployed successfully with it. While they aimed to make the on-campus course and the MOOC “as similar as possible, limitations of the MOOC format meant that there were some differences between the courses.” (Theisen et al., 2016, 4) Apart from the missing face-to-face interactions in the MOOC, which have been substituted by online discussion in a subreddit on reddit.com, there where no group projects designed into the MOOC and the peer review of assignments did not work properly in the MOOC and was finally scrapped due to technical issues with the Google Course Builder platform. And while the on-campus course students where bound to the fixed schedule of class dates, the MOOC students could progress in the course in a relatively self-paced way.

Among the learned lessons from this research are (ibid, 9):

- Preparation & facilitation of MOOCs is very time consuming
- Peer evaluation is challenging in general, and even more so in MOOCs due to increased scaling effects
- If a learning platform only supports multiple choice and true/false questions for assessments, the learning opportunities are limited.
- Google Course Builder and Google Forms had several drawbacks and instabilities and platform choice for MOOCs does matter a lot.

- Informal communication, e.g. in form of videos, throughout the course and tied to current (news) events helps students to engage with the topic.
- When facilitating communication on message boards, timely feedback is of key importance. Additional use of synchronous communication (e.g. webinars) might increase engagement.
- MOOC students - due to often being engaged in work and other projects - tend to underestimate a courses demand on their time, so flexibility in this regard is important.

These points should be kept in mind when integrating secure coding into existing introductory programming courses. As most of them relate to online/distance learning in MOOCs, they also point towards crucial features that could be strengthened in existing on-campus courses, especially if interactivity is not yet explicitly designed into the courses.

A more specific focus on integrating additional materials in on-campus courses was already presented above in (Raina et al., 2015) and (Raina et al., 2016). In general the *Security Injections @ Towson* project seems promising to spread the integration of secure coding issues into (introductory) programming courses. A more detailed view is provided in another 2016 publication out of the same project (Taylor & Kaza, 2016). Their starting point is that, still, "the majority of computing students are not exposed to principles of secure coding. Additionally, most undergraduate security courses are upper level and are offered after students have established coding techniques" (ibid, 2). While it is now a generally acknowledged principle that security should not just be seen as an add-on but be built in from the start, this approach to teach "students to program first and learn secure coding later" (ibid) is a practice that fundamentally contradicts the requirements of today's computer science education.

Besides the necessity of integrating security education early on in computer science education for effective adoption of secure coding practices, they argue, there are additional organisational advantages to integrating secure coding into existing programming and other early CS courses, instead of creating additional elective courses. Because in this way security education can be incrementally integrated by small changes in existing courses. Also the security mindset is established across the curriculum and therefore not tied to a single issue. This also should lead to an increased effectiveness of those elective security topics that are already available. Nevertheless, "effective security integration has significant challenges—a lack of resources, courses that are overcrowded with difficult topics and struggling students, faculty who are untrained in security, and an academic culture that fails to recognize the consequence of software vulnerabilities" (ibid, 4).

The primary objectives for their security injection modules are to increase the security awareness of students and their abilities to apply security principles and secure coding practices as well as to increase the awareness of secure coding concepts among the computer science faculty (ibid, 5), as "many instructors are untrained in developing secure software" (ibid, 14).

Each security injection module consists of the following 5 components:

- background information on the topic, including examples and links to articles
- a “code responsibly” section with tips how to code safely
- lab assignments
- security checklists for how to spot the vulnerability in code
- discussion questions to encourage reflection and engagement with the topic beyond the assignment submission

The modules are designed for stand-alone use and should be applicable with “little or no instructor intervention”, and the “[e]stimated completion time for most modules is 20 to 25 minutes” (ibid, 7). This could also be useful for SCRAP and similar efforts of post-hoc integrations in introductory programming courses.

By organising workshops and training sessions at their own university as well as their partner institutions they facilitated instructor buy-in and adoption of the security injection modules at their institutions. By organising workshops, panels and birds-of-feather sessions at conferences on computer science education they reached out to the broader community and tried to increase awareness for secure coding in computing education contexts.

Additionally they created a *Build-A-Lab* programme, spanning three terms with workshops helping teachers to adopt the existing modules and to create new ones. This lead to a few new modules being developed and integrated in the *Security Injections @ Towson* project.

Finally they also made their content available on a website at http://www.towson.edu/securityinjections. Meanwhile this site might already have changed, and its URI redirects to http://cis1.towson.edu/~cyber4all/index.php/security-injections_home (latest check on 2020-04-04). But unlike many other projects reviewed in this thesis, the site is still available and seems to be under active maintenance. The crucial issue is pointed out by the authors themselves: “Although it is challenging to develop quality content, it is as hard to maintain it, let people know about it, provide access, and get people using it.” (ibid, 15)

A closer look at their workshop listing (Towson University, 2017) tells us that they had constant events until July 2017. Afterwards either the project page did not get updated or they stopped doing the workshops. The latter is quite likely and potentially due to the end of some project funding. An inquiry at the authors might close this open question, but in this case I want to use the likeliness of an ended funding to point to a bigger structural issue:

If the advancement and integration of secure coding into computer science education shall be sustainable, no single project can accomplish this. Rather it needs continuous maintenance and planning for corresponding resource allocation in the educational institutions.

2.4 Related Work

In a 2019 master thesis at the TU Wien, titled “Continuous Security in DevOps environment: Integrating automated Security checks at each stage of continuous deployment pipeline”, Mohammed Jawed analyses security requirements and methods to fulfil them for the different stages in the software development lifecycle (Jawed, 2019).

Regarding the requirements to a securely developed software Jawed relies on the *OWASP Application Security Verification Standard* (ASVS), which guides along the lines of what should be verified and tested (ibid, 19). While Jawed used version 3.0 of the OWASP ASVS, meanwhile version 4.0 is available (OWASP, 2020*a*). Jawed also mentioned along the lines of developers becoming aware and understanding such controls that are defined in standards like the ASVS, that little games like the *OWASP Cornucopia* (OWASP, 2020*b*) or *OWASP Snakes And Ladders* (OWASP, 2020*d*) can help to introduce these issues (Jawed, 2019, 21). For further research it would be interesting to evaluate the use of these games in introductory programming courses that integrate secure coding.

The section on secure coding practices is rather short and only lists several (poorly referenced) standards or guidelines as “Industry best practices” (ibid, 35), among them most notably the *OWASP Secure Coding Practices*, which is available as a checklist, currently in version 2 (OWASP, 2010).

For the SAST part of the toolchain, Jawed lists SonarQube as the F/LOSS tool of choice (Jawed, 2019, 59-60), although they use the proprietary Fortify Static Code Analyser for their own toolchain (ibid, 41). Other F/LOSS tools for the develop and build stages listed there, which are potentially useful to gain additional feedback on secure coding issues are:

- OWASP Dependency Check
- DevSkim
- Phabricator
- CheckSec
- Jenkins

This highlights the broader focus on the whole SDLC of full-fledged software development projects. For SCRAP at this stage they are less applicable. But the use of SonarQube reinforces my own choice for the static analysis tool used in SCRAP. DevSkim on the other hand might be quite useful for the integration into introductory programming courses in the same way the ASIDE/ESIDE approach suggests (Teto et al., 2017) (Whitney et al., 2015) (Zhu et al., 2013). DevSkim also is the “spiritual successor” to Yasca, as Michael Scovetta puts it. But while this makes it more usable for integrating secure coding as a principle in introductory programming courses, being an IDE plugin makes it less useful for direct integration into SCRAP.

A 2016 master thesis in IT Security at the FH Technikum Wien by Christoph Lindmaier focuses on automated security tests for web applications (Lindmaier, 2016). But while it provides a more detailed insight into the vulnerabilities listed in the *OWASP Top 10*, its practical part focuses on dynamic application security testing (DAST), using primarily OWASPs Zed Attack Proxy, while the focus of this research is on SAST tools and how to build a toolchain and web service that can be integrated in existing code submission systems.

3 Research Question

The main research questions of this thesis are:

1. Can we build a toolchain of open source secure code analysis and code vulnerability analysis tools, to analyse solutions to programming exercises and to automatically create qualitative feedback?
2. What are the contextual (that is, organisational and educational) requirements for applying such an approach to gain secure coding awareness among students?

To answer these questions and to build a prototype that can be used in future research to evaluate student responses to the automatic feedback, there are at least the following sub questions that should be answered:

1. What is the current state of secure programming/software engineering education?
2. What open source tools for code analysis are available and useful for this case study?
3. How are those tools best combined to analyse code submitted by introductory programming students?
4. Which patterns can we detect and what feedback can we generate out of it?

Findings generated in this project will be of an exploratory kind, as the limited resources for a master thesis project cannot be used to generate a full-fledged qualitative case study. The aim of the project is:

1. to provide a technological basis for further research into this area, by developing the platform prototype, and
2. to provide some initial insights into how reasonable a further adoption of this approach for introductory programming courses is.

The main target audience is to be found in the academic sector, namely developers and DevOps in higher education institutions as well as academics working on educational technologies and didactic in computer science, especially when it comes to knowledge transfer in the area of programming.

4 Methods

To address the described research questions, two main methods are applied. First, an extensive literature review - described in the next section - is used to gather information on the field of software security education and useful approaches that have already been tested in this regard. Second - described in the section thereafter - an API is developed with a prototype implementation of a web service and a web UI, after an evaluation of available F/LOSS static analysers. The API and the implemented prototype components facilitate a RESTful architectural style to provide a platform that can be extended with and integrated in further research.

4.1 Literature review

This chapter provides insights into my process to review current literature on topics of secure coding and software security education. It also highlights all found relevant materials. While the core literature is evaluated in much more detail in chapter 2 on the current state, subsection 4.1.2 on core and extended literature gives a brief overview of additional findings, that might prove valuable to further and extended research projects.

4.1.1 Process and initial findings

To gain an overview of the field and identify core materials and reference, I chose the following approach to literature search and review:

1. Browse local university library catalogues, namely those of the FH Technikum Wien (FHTW), the TU Wien (TU) and the University of Vienna (UV). Use the search terms “secure code”, “secure coding”, “software security” to identify core materials on secure coding issues. Use the search term “secure code teaching” and “software security education” to identify a subset of materials that deal with issues pertaining to feedback generation in regard to secure coding. Use the search term “computer security” to identify potential standard works that also encompass the topic of secure code.

2. Browse journal databases, namely the Directory of Open Access Journals, the Science Direct College Edition and the IEEE Xplore Digital Library for the same search terms as above. Access to the latter two is provided by FHTW. For Science Direct the *Physical Sciences and Engineering* collection is contained in the access package. For IEEE Xplore it contains the *IEEE/IET Electronic Library*.

3. Browse one of the local university libraries for “php security” and “php secure code”.
4. Do a web search on php security and python security to identify auxiliary resources on the two languages in use in this project.

For the first step I started with the FHTW catalogue. There I found 2 overall hits on “*secure code*” and the same two hits on “*secure coding*”. While one was a book on Java 9, that also deals in how to write secure Java code (Lavieri & Verhas, 2017), the other was a genuine book on writing secure code, which systematically introduces all code-based vulnerabilities and provides language-agnostic strategies how to tackle these, sprinkled with concrete examples in different languages (Howard & Le Blanc, 2003). As it turned out later throughout the literature search, this is one of the few resources dealing with secure code in its core. Although the second edition was from 2003 and there is only a 2009 reprint of this second edition, a preliminary skimming of the books content showed that it is a highly relevant reference on the topic of secure coding.

For the further search of the FHTW catalogue, I continued with the search term “*software security*”. This led to 30 hits. 14 of them are e-books among which there was 1 guideline on software development for security-critical areas (Kriha & Schmitz, 2008), detailing the different components and their management in secure software development, but not specifically about secure code. Also there was 1 practical handbook on security of web applications (Rohr, 2018). The other 12 e-books did not specifically deal with secure software development. There also were 7 theses among the search hits, 2 of which contained security analyses of specific software. None of them were specifically about secure coding. The other 9 hits all have been books, only one of which was about secure coding and secure software development, namely (Howard & Le Blanc, 2003), which I already identified in the search for “secure code”.

As the search terms “*secure code teaching*” and “*software security education*” are subsets of the searches already conducted, this yielded no new results.

Searching for “*computer security*” in the FHTW catalogue yielded 127 hits, with the oldest on cryptology from 1985 (Horster, 1985). 70 hits dated between 2012-2019. The two most relevant of these were (Stallings & Brown, 2015) and (Conklin et al., 2016), 2 standard handbooks on computer security, also containing chapters on secure code and secure software development.

The reason for these relatively small number of overall hits is that the FHTW library catalogue does not contain papers and conference proceedings. But the FHTW provides access to Science Direct and the IEEE Xplore Digital Library, which I used later for identifying more materials. But before going into these libraries, I continued with searches at TU Wien and University of Vienna.

The statistics laid out in tables 5 and 6 was obtained on 2019-07-22 by browsing the library catalogues at https://catalogplus.tuwien.ac.at and https://usearch.univie.ac.at, showing the number of hits in these catalogues for the different search terms.

In both libraries nearly all references I found are also available online through the library system. While the TU has in total much more hits, a substantial part of them is comprised of

Table 5: Results of TU Library Search.

Search Term	Hits	Peer-reviewed	Books	News-paper	Conf
secure code	219244	61417 (28%)	7702 (4%)	79815 (36%)	13921 (6%)
secure coding	49301	24197 (49%)	1494 (3%)	11349 (23%)	5264 (11%)
software security	1080952	159977 (15%)	13247 (1%)	563940 (52%)	53801 (5%)
secure code teaching	14084	7204 (51%)	2670 (19%))	806 (6%)	1257 (9%)
software security education	161074	51197 (32%)	3991 (2%)	45348 (28%)	8899 (6%)
computer security	815960	179495 (22%)	20752 (3%)	252978 (31%)	87331 (11%)

Table 6: Results of UV Library Search.

Search Term	Hits	Peer-reviewed	Books	News-paper	Conf
secure code	154096	120260 (78%)	6052 (4%)	1066 (1%)	13712 (9%)
secure coding	63865	54235 (85%)	789 (1%)	254 (0%)	5006 (8%)
software security	321630	219250 (68%)	7522 (2%)	3172 (1%)	54735 (17%)
secure code teaching	26655	21814 (82%)	2426 (9%)	137 (1%)	905 (3%)
software security education	99179	79699 (80%)	1661 (2%)	1237 (1%)	8238 (8%)
computer security	424194	250574 (59%)	14822 (35%)	4575 (1%)	89305 (21%)

newspaper articles, while at the UV the majority is comprised of peer-reviewed journals and conference proceedings (together making more than 80% of the total hits).

The difference in the two result sets is also displayed in figures 1 and 2. From these two distributions there is not much to conclude, except that apart from the massive collection of newspaper references in the TU catalogue, both university libraries seem to hold similar amounts and distributions of search results regarding the 6 search terms. From the comparison of peer-reviewed papers, conference proceedings, and books respectively between the two libraries, as shown in figures 3, 4, and 5 can be assumed that similar sources are available when it comes to conference proceedings, but for peer-reviewed papers and books the TU library catalogue could provided a slightly broader perspective on the topic. Therefore I used the TU library catalogue as a primary source to identify relevant materials. The UV library catalogue was used later to do an auxiliary check if the TU results sufficiently covers the issues of secure coding and issues of teaching and feedbacking on (in)secure code.

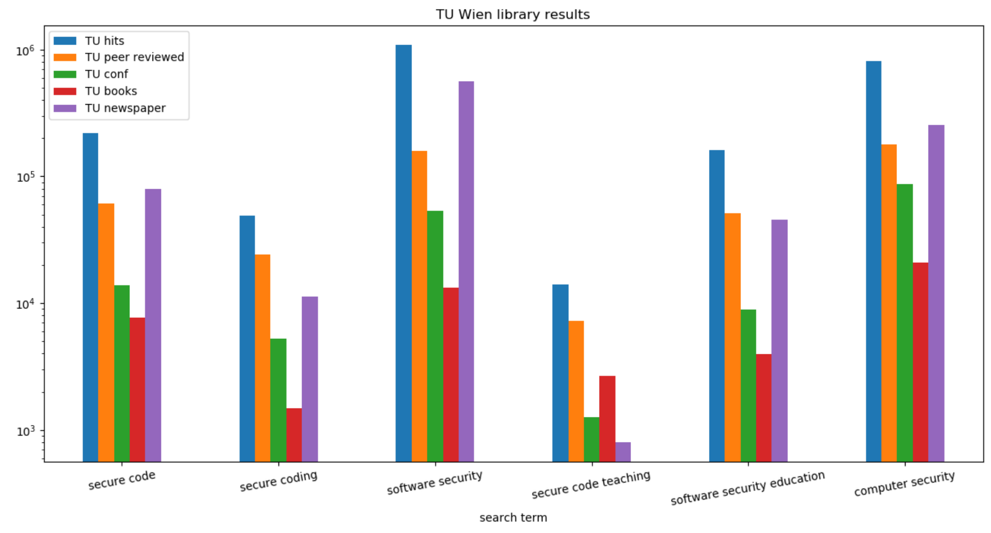


Figure 1: Logarithmic distribution of search results in the TU library catalogue.

In order to identify relevant materials from the TU library in context of this master thesis project, I had to drastically reduce the amount of results. Therefore I applied two filters for all search terms:

- Publication date between 2012 and 2019 (inclusive)
- Order by relevancy and use the first 100 hits

These results were screened based on titles first. Those which seemed relevant I further screened based on abstracts, summaries and table of contents where available.

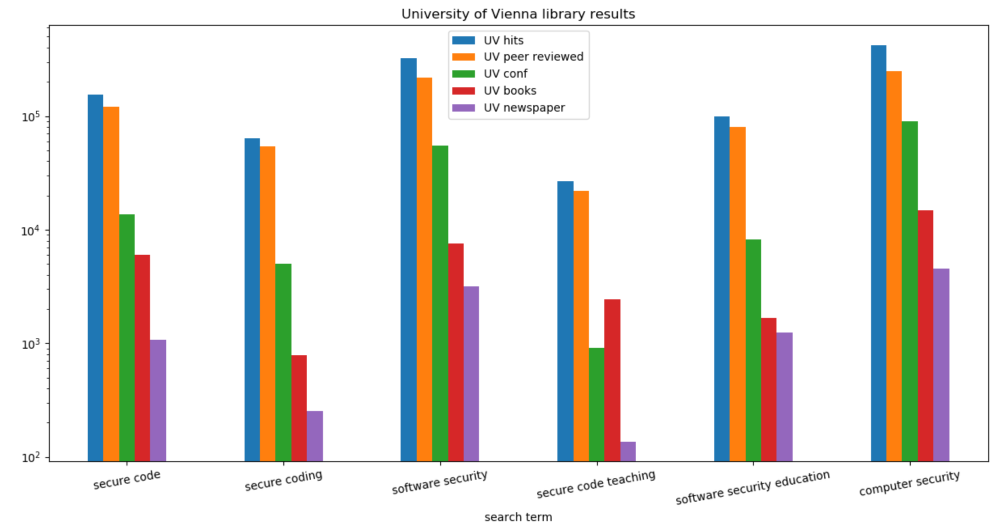


Figure 2: Logarithmic distribution of search results in the UV library catalogue.

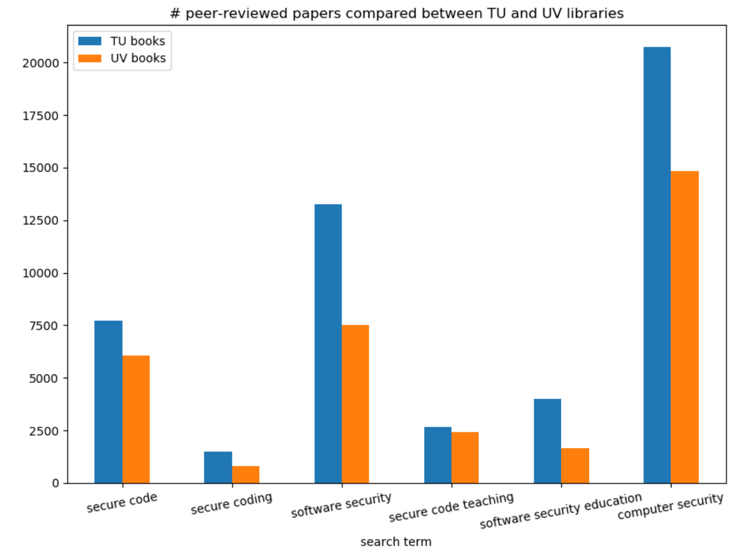


Figure 3: Comparison of peer-reviewed search results between TU and UV.

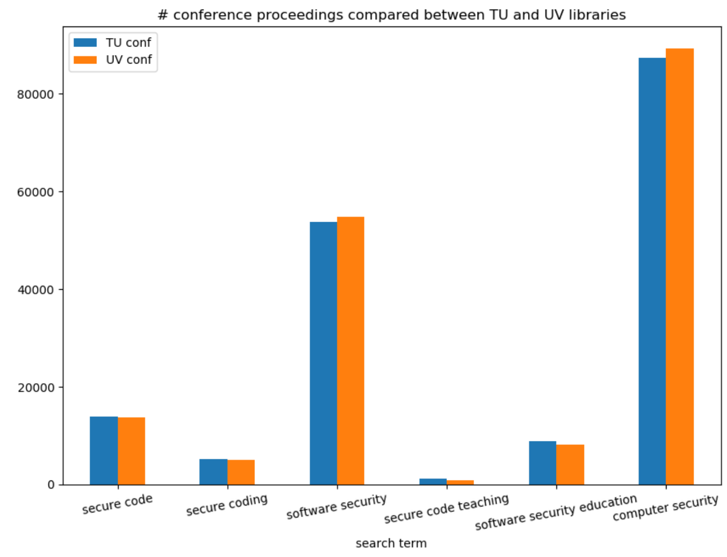


Figure 4: Comparison of conference search results between TU and UV.

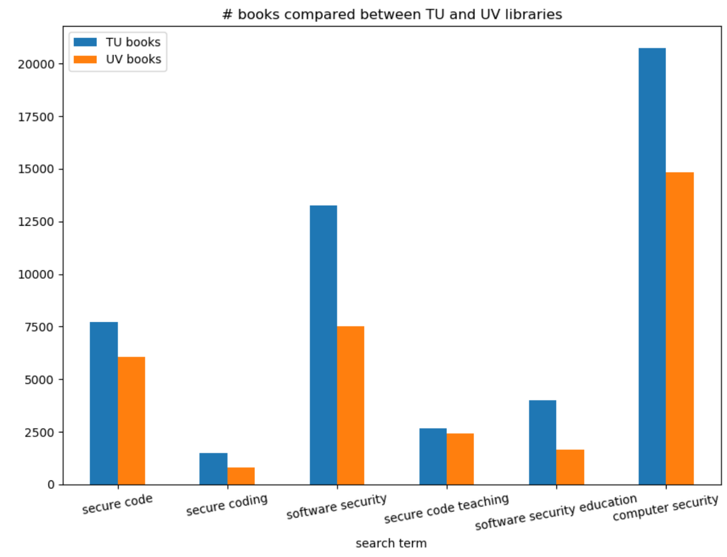


Figure 5: Comparison of books search results between TU and UV.

4.1.2 Core and extended literature

First I started with “secure code teaching” and “software security education”, as these are the smallest result sets. For “*secure code teaching*” the search for books yielded no relevant material.

The search for peer-reviewed journal provided only one paper among the first 100 hits that came close (Popa, 2012), but it deals mostly in securing the software development life cycle and software assurance. What proved more promising was to use an additional filter “Teaching Methods” as topic. This reduced the overall result set to 65 entries.

While not directly on issues of secure coding, (Sorva, 2013) describes struggles of students in introductory programming courses. They provide insights that might also be valuable for adopting secure coding practices.

Also not directly on secure coding, but on social representations of cybersecurity by students in introductory information systems courses, (Pawlowski & Jung, 2015) provide some suggestions for approaches to instructional design that help raise the awareness of cybersecurity.

On the aspect of game-based learning (Giannakas et al., 2018) provide a critical review of publications on mobile game-based learning from 2004 to 2016. Although this is not a core topic of my thesis, it provides some thoughts on the potential gamification of SCRAP.

Removing the publication date filter additionally revealed (Bishop, 2006) as potentially relevant auxiliary material.

When searching for conference papers with the search term *secure code teaching* between 2012 and 2019, several relevant references could be identified.

(Chi et al., 2013) describe modules they built “for teaching secure coding practices to STEM students” and provide some evaluation results.

(Raina et al., 2014) assess “modules for CS0, CS1, CS2 and Computer Literacy courses that target key secure coding concepts including integer overflow, buffer overflow, and input validation”. While this paper’s full text is not accessible through the TU library catalogue, the ACM Digital Library listed (Kaza et al., 2015) and (Kaza & Taylor, 2018) as additional resources. All of them provide more references with context information on the state of the field. Additionally (Raina et al., 2016) is listed in the TU library and accessible as full text, describing a study with 53 students as participants.

(Zhu et al., 2013) developed a prototype tool called “ASIDE” for educating students on secure coding. The tool was developed as an IDE integration. Apart from that the aim to educate students on secure coding along their programming exercises and to evaluate this tool is similar to my own approach.

(Whitney et al., 2015) is an additional report on their project, describing a field study on their tool applied to a Web programming course.

(Xie et al., 2015) is a proposal to utilize Microsoft’s Code Hunt platform in order to increase secure coding skills among programmers.

(Taylor et al., 2013) describes a panel session at the 44th ACM technical symposium on computer science education tackling “the myths and realities” of teaching secure coding.

(Weir et al., 2016) describe how game and story telling aspects can be used in programming education to foster secure coding practices, with a focus on mobile apps.

(Schuckert et al., 2017) examine SQL injections in PHP code and if and how source code patterns changed over time, concluding that "developers had software security in mind, but nevertheless created vulnerabilities".

Based on a case study of Cross-Site Scripting attacks (Teto et al., 2017) argue for the need to better integrate Defensive Programming into computer science courses.

(Hooshangi et al., 2015) analysed the quality of coding exercises in an introductory security course. One of their results was that “students who learn to write good defensive programs can write effective attack programs, but the converse is not true”.

Another study highlighting the value of defensive programming is based on student exercises where students had to develop reference monitors in Python, on which other student’s attack code should be run (Cappos & Weiss, 2014).

Based on teaching an introductory programming course to students for four years (Pournaghshband, 2013) argues for the value of including the security mindset early on in computer science education.

Outlining two models of an introductory course to cryptography and computer science in a traditional liberal arts college, (Buchele, 2013) provides context and ideas on how to make computer security knowledge more accessible.

In an analysis and comparison of software security patterns to design patterns, (Bunke, 2015) provide some insights for the usefulness of software security patterns.

Moving on to the search for *software security education* revealed several of the findings from the previous search, but it also brought to screen several additional resources, that are highly relevant for this work.

(Bishop et al., 2015) is not a monograph or textbook but, as conference proceedings, came up in the search filtered for books. It includes an exclusive section on *Software Security Education* and can be treated as a recent core reference, especially (van Niekerk & Futcher, 2015), (Raina et al., 2015), and (Jøsang et al., 2015).

(Jawed, 2019) is a TU master thesis on integrating automated security checks in all stages of a CD pipeline. As SCRAP could be seen as one stage of such a pipeline, this reference might yield some insights into how to optimize the code analysis.

Somewhat related is (Kernegger, 2013), also a TU master thesis, investigating how to improve the error detection rate with automated security testing tools. But due to its focus on testing instead of code analysis, I only use it as a peripheral source for context.

Among the highly relevant papers found are (Taylor & Kaza, 2016), which relates to the same research projects as (Raina et al., 2016), (Kaza et al., 2015) and (Kaza & Taylor, 2018)

In the search results for *software security education* and filtered by conference proceedings, I encountered several references, already seen in other searches before. Also there where several works on general aspects of information & software security education, which I filtered out, because they did not focus on secure code specifically.

(Jøsang et al., 2015) argue that while there is industry agreement that security should be part of the whole software development life cycle, there is much less focus on making security part of software development education. They understand software development education that does not explicitly address security and secure coding as a major contributor to the software vulnerability landscape.

(Uskov, 2013*b*) reports on design, implementation and evaluation of the 'state-of-the-art undergraduate and graduate curriculum and courseware' in software and web application security (SWAS) at Bradley University. (Uskov, 2013*a*) is a more extensive version on the same project.

(Theisen et al., 2016) report on a comparison of software security coursework in an on-campus classroom and a Massively Open Online Course (MOOC) version.

(Pancho-Festin & Mendoza, 2014) present their learning from integrating security topics into undergraduate software engineering courses, with a special focus on web application security.

Moving on to the search for *secure code* and *secure coding*, to reduce the relevant resource set to a reasonable amount in context of this master thesis, I used a *logical and* operator for the terms *secure*, *code*, and *coding*, which reduced the overall results to 22.976 hits in the TU library catalogue, which in turn filtered by years from 2012 onwards resulted in 14.511 hits. These have been filtered then for books (582 hits), papers (6.353 hits) and conference proceedings (1.775 hits) respectively.

A book on software quality assurance by (Walkinshaw, 2017) contains sections about testing and code safety, that might provide insights into the state of secure coding in industry.

(Sahu & Tomar, 2017) analyse web application code and develop a graph-based interactive prototype system to help developers follow secure coding standards.

(Kaza et al., 2018) is another report on Towson University's Security Injections project, which came up several times in the search for software security education. In this short article they reference the project page at https://www.towson.edu/securityinjections, which holds valuable resources in form of short modules teaching secure coding concepts, which can be injected in more classical computer science courses.

(Nembhard et al., 2019) develop and evaluate a recommender system for Java code to find and remediate security vulnerabilities. The recommender system approach is not directly relevant for the SCRAP prototype, but it might be interesting to evaluate the use of a recommender system as an add-on in future research.

(Anis et al., 2018) present a "system that helps developers to implement security measures on the client side code based on the best practices of secure coding" by developing an integrity verification module for JavaScript-based applications.

(Rahaman et al., 2018) propose a "90-minute tutorial to teach participants the principles and practices of Java secure coding", focusing on API misuse and introducing a tool to automatically detect API misuse in Java code.

(Meng et al., 2018) identify a "huge gap between security theory and coding practices" by analysing *StackOverflow* posts related to Java and secure coding.

(Heymann & Miller, 2018) describe a tutorial on secure coding practices and assessment tools. In this course they also present “SWAMP”, a free and open “facility to provide access to a large collection of tools for a variety of languages and environments”.

Given the extensive results of the searches so far, I limited the search for *software security* to the first 50 results filtered for books from 2012 onwards, that do not deal with the management of software security or a secure development life cycle, but focus primarily on providing a technical overview of issues for secure software development.

For the last and most general search term, *computer security*, I only conducted a search for books, given the extensive results already produced by the other searches. Additionally I filtered for printed books, as the result set without this filter mostly contained conference proceedings. Although the result set was much more extensive than for the same search at the FHTW, the search yielded the same two relevant results, only with a greater variety on the available editions (Stallings & Brown, 2018) (Conklin et al., 2016).

Regarding the search term *php secure code* there wos no hit in the FHTW library and 1 hit for *php security*, a handbook on security issues in PHP and how to solve them with secure coding patterns and approaches in PHP (Snyder et al., 2010). As its publishing date is in 2010, it is referring to PHP 5.3, so this could be counted more as a historical back reference.

In a web search on *python security*, one notable article describing *10 common security gotchas in Python and how to avoid them* could be found (@anthonypjshaw, 2018).

From all of the above the around half of the resources have been classified as core materials and are reviewed in more detail in chapter 2, as listed in footnote [1].

A notable additional source that I found in the core literature, is (Stivalet & Fong, 2016). This would certainly be valuable core material for future research and a thorough evaluation of an analysis platform, as they provide a rigid test case generator. But due to the scope of this work and the already available vulnerability code through *DVWA*, this reference is categorised as auxiliary material nevertheless.

All other material was classified as further auxiliary material that could not be reviewed in detail, but should be evaluated for further research. They are listed in footnote [2].

[1]Core literature: (Howard & Le Blanc, 2003) (Sorva, 2013) (Chi et al., 2013) (Uskov, 2013*b*) (Uskov, 2013*a*) (Zhu et al., 2013) (Pancho-Festin & Mendoza, 2014) (Hooshangi et al., 2015) (van Niekerk & Futcher, 2015) (Raina et al., 2015) (Jøsang et al., 2015) (Whitney et al., 2015) (Conklin et al., 2016) (Raina et al., 2016) (Taylor & Kaza, 2016) (Theisen et al., 2016) (Schuckert et al., 2017) (Teto et al., 2017) (Sahu & Tomar, 2017) (Anis et al., 2018) (Heymann & Miller, 2018) (Kaza et al., 2018) (Jawed, 2019)

[2]Auxiliary literature: (Bishop, 2006) (Snyder et al., 2010) (Buchele, 2013) (Kernegger, 2013) (Pournaghshband, 2013) (Taylor et al., 2013) (Cappos & Weiss, 2014) (Bunke, 2015) (**?**) (Pawlowski & Jung, 2015) (Xie et al., 2015) (Weir et al., 2016) (Walkinshaw, 2017) (Giannakas et al., 2018) (Meng et al., 2018) (Rahaman et al., 2018) (Rohr, 2018) (Nembhard et al., 2019)

As the search conducted so far already yielded substantial results, several of them retrieved through the TU library's access to IEEE Xplore and the ACM Digital Library, I did not conduct an additional systematic search in the IEEE Xplore and Science Direct catalogues provided by the FHTW library. A further screening of these databases would certainly make sense for a larger scale follow up project.

4.2 Prototyping RESTful Webservices

The development of the SCRAP prototype follows the REST architectural style, as it was proposed already in 2000 by Roy Fielding in chapter 5 of their dissertation (Fielding, 2000), or rather as what nowadays is more broadly known as a RESTful web service, which is nicely summarized in the REST API Tutorial (restfulapi.net, 2020).

One reason for this approach ist that already at the dawn of the REST architectural style in the beginning of the current millennium, the value of web services for distributed application systems was more and more increasing (Alonso et al., 2004). And while for some time SOAP was still the primary way to implement web services, meanwhile REST is the major approach to designing and implementing web services, especially when it comes to services that are not only used within a single organisation but that can or should be exposed to third parties or the public (Lange, 2016).

And while the W3C Working Group in 2004 still wrote that the purpose of "REST-compliant Web services" is "to manipulate XML representations of Web resources using a uniform set of 'stateless' operations" (W3C Working Group, 2004), today the JSON format (short for JavaScript Object Notation) is widely used in RESTful web services.

The advantage of the RESTful approach in combination with using JSON as the data exchange format is that it lends itself very much to rapid prototyping. For one, there is no need to use any other underlying technology than HTTP, which is already available in any browser, can be accessed by simple `curl` calls from the command line and is not only easy to parse for computers but also quite readable for humans, in that case particularly developers.

For these reasons I have chosen to implement a RESTful web services, which can be quickly prototyped, easily extended and also integrated into any other system that is capable of speaking HTTP to the server where the SCRAP web service resides.

It is beyond the scope of this thesis to also introduce the RESTful approach in detail, but especially for interested developers the above mentioned REST API Tutorial and Kenneth Lange's Little Book on REST Services (Lange, 2016) are good and compact resources to dig into the topic. But even without a deep understanding of REST and the principles of RESTful web services, the approach will explain itself through the description of the prototype in the following chapter. A basic understanding of HTTP should be enough to follow what is going on, which is another reason why RESTful web services are so successful, although not every RESTful webservice does strictly adhere to all REST principles.

5 Prototype

5.1 Design

A lot of the architectural design of SCRAP is made explicit in the following section 5.2 on the implementation of the SCRAP prototype. Here I want to take a quick look on some of the key requirements for such a platform, if it should provide any added value in actual introductory programming contexts.

This is of course a preliminary perspective, which would have to be extended and made more concrete in context of an actual organisation that wants to adopt such an approach. This is discussed in more detail in section 5.3 on the evaluation of the prototype and in the final conclusion chapter 7.

The key requirements which can be formulated from the inputs of the literature review and my own experiences in introductory programming contexts (as a learner as well as a teacher and facilitator), are as follows:

- The system should provide a core server component that is easily accessible from different clients and can be integrated through a well documented API into other services that already handle code submissions.
- There should be a lightweight web client available, which can be used additionally to the integration into other code submission systems, so that students can explore their code vulnerabilities on their own and not just post-hoc to their code submissions.
- When providing the results of a scan and information on the issues found it is important to make transparent how the code was scanned and how users could apply the scan on their own
- The technical system should be well documented, in a way that different institutions can evaluate and integrate it into their own contexts. Otherwise there is a strong likelihood that the project code grows old and at some point becomes one of the many originally promising but now unmaintained and defunct projects to foster software security education - or in other words: SCRAP might quickly become scrap, if extendability and accessibility through good documentation are not provided.

On the matter of requirements, we also have to face the fact that when it comes to secure coding we need not only the functional requirements that have to be implemented but also explicit “security requirements [which] need to be both specific and positive” (Conklin et al., 2016, Ch. 18, Section: Requirements Phase). As we cannot assume such specific requirements can be formulated in context of introductory programming courses, the generated feedback of the prototype system has to be formulated accordingly. We cannot assume that students had sufficient exposure to secure coding issues. For all instances of potentially vulnerable code, that our toolchain finds, we have to provide general information and context of the vulnerabilities as well as examples of secure code that solves the same thing.

Regarding the rule sets for scanners and the explanations of vulnerabilities there certainly has to be a major effort to generate new content for a productive use of SCRAP. In terms of the prototype I can only provide an example on how rules and explanations can be added. But if an approach as presented with SCRAP should be successfully adopted in actual educational organisations, the major effort will not be a technical one to implement the prototype into a fully functioning platform, but an educational one, drawing from the resources like found in the literature review, to generate good explanations and to adopt the scanner rule sets to the actual code submissions in introductory web application programming courses.

The CWE provides a slice view on Weaknesses in Software Written in PHP, listing many common weaknesses in PHP code, including descriptions and potential ways to mitigate the problem. This resource could be used for the creation of additional scanner rules and explanations.

But certainly this approach does not have to be limited to PHP code. Especially when it comes to available F/LOSS scanners, which can be integrated into SCRAP, there might be even better opportunities for languages as C/C++ or Java, which are still very common for introductory programming courses in the early semesters of computer science programmes, before students can even peek into web application programming.

5.2 Implementation

5.2.1 REST API

My main design logic is derived from best practices in RESTful API design, as they are highlighted in an article on Best Practices in API Design by Swagger (swagger.io, 2020). Their three key principles are to be *easy to read and work with*, to be *hard to misuse* and to be *complete and concise*. For a project with the scope of a master thesis providing a functional prototype on top of extensive literature research this means, that the API should also only focus on the most relevant parts needed.

When it comes to the resources which are exposed through the API, then we only have three core resources:

- **Scans**: those are the actual scans of submitted code
- **Scanners**: represent the different tools that are used by scans to find issues

- **Explanations**: descriptions of vulnerabilities and mitigation strategies, which might be referenced by scans in relation to the found issues

Beyond those three key resources there are also **Files** and **Issues**, which should be accessible as resources. Files are the actual files that have been submitted for the scans and issues are the one or more issues a scan did find on those submitted files. The issues then could contain references to explanations, which should help the user to understand and mitigate the security issue at hand.

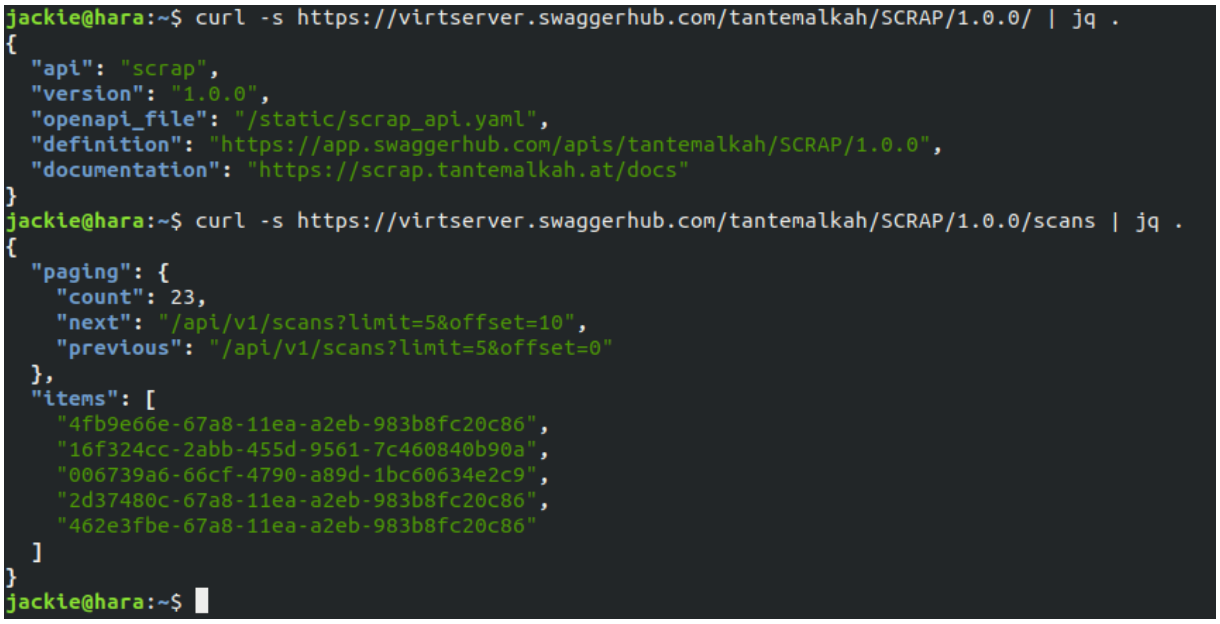
This leads to the concrete API design which was encoded in a YAML format and adheres to the OpenAPI 3.0.0 specification (Miller et al., 2020). The actual scrap_api.yaml file is part of the scrap-api-server repository on GitLab, which is also available on the accompanying data disc. The file's content is also attached as a listing in the appendix. A marked up representation (and documentation) of the API can be accessed through a link on the SCRAP project site at https://scrap.tantemalkah.at, or directly on SwaggerHub, where it was initially created. While SwaggerHub only offers limited options for free accounts, the SwaggerEditor and the Swagger UI are F/LOSS tools which can be used in the same manner locally to create an OpenAPI conformant API description and to generate a visually marked up documentation of it.

While the full specification of the SCRAP API is linked to and attached, for brevity listing 2 provides just the available paths and methods of the API with the summary of what it does:

```
paths:
  /:
    get:
      summary: Retrieve the server and API meta information
  /scans:
    get:
      summary: List all available scans of a user
    post:
      summary: Submit a new scan
  /scans/{id}:
    get:
      summary: Retrieve meta information for a single scan
    delete:
      summary: Delete a single scan
  /scans/{id}/files:
    get:
      summary: Receive listing of all files of a scan
  /scans/{id}/files/{filepath}:
    get:
      summary: Retrieve a single file from a scan
  /scans/{id}/blob/{filepath}:
    get:
      summary: Receive a single file from a scan
  /scans/{id}/issues:
    get:
      summary: Receive a listing of all issues found in a scan
  /scans/{id}/issues/{issueid}:
    get:
      summary: Receive a single issue from a scan
  /explanations:
    get:
      summary: Get a list of available explanations
  /explanations/{slug}:
    get:
      summary: Retrieve an explanation to a vulnerability
  /scanners:
    get:
      summary: Retrieve a list of available scanners
```

Code 2: Listing of the available paths and methods in the SCRAP API

One advantage of using SwaggerHub instead of the stand-alone Swagger Editor and Swagger UI is, that it comes with an automatically configured mocking server, which can be used to test the API at specification time. Even right now (if SwaggerHub still exists and did not change its policy in a major way), you should be able to click the link to the mocking server and get some meaningful results, if your browser does know how to render JSON responses well (as for example *Mozilla Firefox* does). You can achieve the same results with a simple `curl` call, as depicted in figure 6.



```
jackie@hara:~$ curl -s https://virtserver.swaggerhub.com/tantemalkah/SCRAP/1.0.0/ | jq .
{
  "api": "scrap",
  "version": "1.0.0",
  "openapi_file": "/static/scrap_api.yaml",
  "definition": "https://app.swaggerhub.com/apis/tantemalkah/SCRAP/1.0.0",
  "documentation": "https://scrap.tantemalkah.at/docs"
}
jackie@hara:~$ curl -s https://virtserver.swaggerhub.com/tantemalkah/SCRAP/1.0.0/scans | jq .
{
  "paging": {
    "count": 23,
    "next": "/api/v1/scans?limit=5&offset=10",
    "previous": "/api/v1/scans?limit=5&offset=0"
  },
  "items": [
    "4fb9e66e-67a8-11ea-a2eb-983b8fc20c86",
    "16f324cc-2abb-455d-9561-7c460840b90a",
    "006739a6-66cf-4790-a89d-1bc60634e2c9",
    "2d37480c-67a8-11ea-a2eb-983b8fc20c86",
    "462e3fbe-67a8-11ea-a2eb-983b8fc20c86"
  ]
}
jackie@hara:~$
```

Figure 6: Accessing the SCRAP API on the SwaggerHub mocking server with curl

While the API documentation on http://scrap.tantemalkah.at/api-doc/ describes the use of all API endpoints and also provides code samples for several programming languages, the API specification on https://app.swaggerhub.com/apis/tantemalkah/SCRAP/1.0.0 allows to view the YAML source alongside a nicely rendered description of all API endpoints, parameters and schemas used in the specification.

5.2.2 API server

A prototypical implementation of the SCRAP API, as described in the previous section, was done in *Python*, using the *Flask* framework and its *Flask-RESTful* extension. The whole code can be accessed in the scrap-api-server git repository on GitLab as well as a copy of it on the accompanying data disc. The setup and how to clone the repository are described in the repository's *README.md* file.

The development environment was a TUXEDO InfinityBook Pro 13 v3 notebook with Ubuntu 18.04 and Python 3.6 in a *virtualenv* environment with all the dependencies listed in and installable with `pip` from the *requirements.txt* file in the repo's root directory. The main requirements to run the server are *Flask*, *Flask-RESTful*, *Flask-Cors*, *mysql-connector*, *PyYAML* and *yara-python*. As a database *MariaDB* was chosen, which at the time was available from the Ubuntu repository in version 10.1.44.

All files from a fresh clone of the repository (except for the .git directory) are presented in listing 3, which has been generated with the command `tree -a -I .git --charset=ascii` inside the repo's root directory:

```
.
|-- common
|   |-- auth.py
|   |-- default_responses.py
|   |-- __init__.py
|   `-- sanitize.py
|-- config.sample.py
|-- db_init.sql
|-- db.py
|-- .gitignore
|-- __init__.py
|-- LICENSE
|-- README.md
|-- requirements.txt
|-- resources
|   |-- explanation.py
|   |-- explanations
|   |   |-- placeholder1.yaml
|   |   |-- placeholder2.yaml
|   |   |-- README.md
|   |   |-- sqli_unsanitized_id.yaml
|   |   |-- _template.yaml.sample
|   |   `-- yara.SQLi.yaml
|   |-- files.py
|   |-- index.py
|   |-- __init__.py
|   |-- issues.py
|   |-- scanner.py
|   `-- scan.py
|-- scrap.py
```

```
`-- static
    `-- scrap_api.yaml

4 directories, 27 files
```

Code 3: Listing of the scrap–api–server repo's contents

For a close look into all files I refer to the online scrap-api-server repository or it's copy on the accompanying data disc.

I will walk through some of the core components here. The main file to start the server is */scrap.py*, which sets up the Flask app and initialises basic configuration settings, taken from */config.py* (which has to be set up first, based on the template in */config.sample.py*). It also adds all the routes to the Flask app, which correspond to the paths presented in the API section above. This is done by the code excerpt in listing 4:

```
api.add_resource(Index, '/')
api.add_resource(ListOfScans, '/scans')
api.add_resource(Scan, '/scans/<string:id>')
api.add_resource(ListOfFiles, '/scans/<string:scanid>/files')
api.add_resource(Issue, '/scans/<string:scan_uuid>/issues/<int:issue_id>')
api.add_resource(ListOfIssues, '/scans/<string:scan_uuid>/issues')
api.add_resource(File, '/scans/<string:scanid>/files/<path:path>')
api.add_resource(FileBlob, '/scans/<string:scanid>/blob/<path:path>')
api.add_resource(ListOfScanners, '/scanners')
api.add_resource(ListOfExplanations, '/explanations')
api.add_resource(Explanation, '/explanations/<string:slug>')
```

Code 4: Adding the routes to the Flask app

The classes that are handling the API requests, which are tied to the routes through the `api.add_resource` methods in the above listing, are all imported from the files in the */resources* folder.

The most basic version of a resource handler is the `Index` resource class in */resources/index.py*, and displayed in listing 5:

```
class Index(Resource):
    def get(self):
        return {
            'api': 'scrap',
            'version': '1.0.0',
            'openapi_file': url_for('static', filename='scrap_api.yaml'),
            'definition':
                'https://app.swaggerhub.com/apis/tantemalkah/SCRAP/1.0.0',
            'documentation': 'https://scrap.tantemalkah.at',
        }
```

Code 5: The Index resource of the server

For this simple endpoint we only need to define a `get` method, which handles incoming HTTP GET requests and just returns a dictionary, which is translated by Flaks-RESTful to a JSON object. While the classes in */resources/explanation.py*, */resources/files.py* and */resources/issues.py* follow the same simple principle, only with some added application logic to handle request parameters and retrieve the data from the database or parse the YAML files in the */resources/explanations* folder, the classes in */resources/scan.py* and */resources/scanner.py* are of more interest, as they handle also the scan submission and the analysis of the scans. These will be explained in more detail in the next section.

Files in the code base that have not been mentioned so far are the files in the */common* folder, which provide basic authentication and sanitization as well as a set of default responses for the request handlers. In the */static* folder resides only the *scrap_api.yaml* file containing the API specification, which is linked to in the response from the `Index` resource. Finally the *db.py* file contains all functions to read from and store to the database.

5.2.3 Scanner integration

For the scanners and the resulting scans to be accessed, the `ListOfScans` and the `Scan` resource classes handle GET and POST requests to the `/scans` and GET and DELETE requests to the `/scans/<string:id>` endpoints respectively.

The method for returning a list of scan is quite succinct, also because the prototype does not implement paging, as seen in listing 6:

```
class ListOfScans(Resource):
    def get(self):
        if not auth.isUser(request.headers):
            return resp.NotAuthorized
        items = db.queryScans(request.headers.get('X-API-KEY'))
        response = {
            'paging': {
                'count': len(items),
                'next': '',
                'previous': '',
            },
            'items': items
        }
        return response
```

Code 6: Method to handle a request for a list of scans (in scan.py)

Also the methods for returning a scan result and to delete it are quite succinct, as listing 7 shows:

```
class Scan(Resource):
    def get(self, id):
        if not auth.isUser(request.headers):
            return resp.NotAuthorized
        if not sanitize.isUuid(id):
            return resp.InvalidUuid
        response = db.getScan(id, request.headers.get('X-API-KEY'))
        if not response:
            return resp.ScanNotFound
        return response

    def delete(self, id):
        if not auth.isUser(request.headers):
            return resp.NotAuthorized
        if not sanitize.isUuid(id):
            return resp.InvalidUuid
        if request.headers.get('X-API-USER') == current_app.config['PUBLICUSER'] \
            and not current_app.config['PUBLICDELETE']:
                return resp.NoPublicDelete
        if not db.deleteScan(id, request.headers.get('X-API-KEY')):
            return resp.ScanNotFound
        return None, 204
```

Code 7: Methods to handle requests for a single scan and to delete it (in scan.py)

Most of the logic is needed for the `post` method of the `ListOfScans` resource, as it handles the upload of new files and archives which should be scanned, and then initiates the scan. The following listing 8 shows the handling of the request parameters and the processing of the uploaded files:

```
def post(self):
    if not auth.isUser(request.headers):
        return resp.NotAuthorized
    if not re.match(r'^multipart\/form-data(; boundary=[-]*[0-9]*)?$',
        str(request.content_type)):
        return resp.UseMultipart
    if not 'file' in request.files:
        return resp.FileNeeded
    file = request.files['file']
    if not '.' in file.filename:
        return resp.WrongFileType
    suffix = file.filename.rsplit('.', 1)[1]
    if not suffix in current_app.config['UPLOADS']['allowed_types']:
        return resp.WrongFileType
    file.seek(0, os.SEEK_END)
    filesize = file.tell()
    file.seek(0, os.SEEK_SET)
    if request.headers.get('X-API-KEY') == current_app.config['PUBLICUSER'] \
        and filesize > current_app.config['UPLOADS']['public_size_limit']:
            return resp.FileTooBig
    if filesize == 0:
        return resp.NoEmptyFiles

    # after this first validation we store the file in a temporary folder
    # to do further checks on the file/archive
    dn_temp = os.path.join(
        current_app.config['UPLOADS']['temp_folder'],
        str(uuid.uuid1())  # if someone else submits the same filename while
            still processing this one
        )
    fn_temp = secure_filename(file.filename)
    os.mkdir(dn_temp)
    file.save(os.path.join(dn_temp, fn_temp))

    # if it is not a single file, but a tar archive we have to check for
    # potential path traversals
    if suffix == 'tgz':
        archive = tarfile.open(os.path.join(dn_temp, fn_temp), 'r:gz')
        contains_php = False
        for f in archive.getmembers():
            if f.name.startswith(('../', './', '/')):
                return resp.InvalidArchiveContent
            if f.name.endswith('.php'):
                contains_php = True
        if not contains_php:
            return resp.InvalidArchiveContent
```

Code 8: First part of the ListOfScans.post method (in scan.py), validating the request data and the uploaded files

In the above code section there is nothing particularly noteworthy except that thorough input validation is done, including the handling of the gzipped tar archives. This prototype version only specifically allows for .php and .tgz files to be uploaded, although the configuration files suggests that the allowed file types can be easily extended. A production implementation would certainly have to provide for that and should probably move the code for the file upload handling in to a file in the */common* folder.

Before the actual scan can be started the next section in the `post` method, as provided in listing 9, generates an initial scan entry in the database to retrieve its UUID, and only then moves the uploaded files to their final destination and populates the *files* table in the database with corresponding entries:

```
# ok, we are good. we can store/extract the files to their final
# location. but need to generate the new scan in the DB first,
# to get its UUID, which we use for the storage path
scan = db.submitScan(request.headers.get('X-API-KEY'))
scan_uuid = scan['id']
# TODO for post-prototype stage:
# implement multithreaded version; the response could be returned now
# while the file indexing and the acutal scans can be started independently
# to update the scan (and its progress) in the DB
dn_final = os.path.join(current_app.config['UPLOADS']['folder'], scan_uuid)
os.mkdir(dn_final)
if suffix == 'php':
    os.rename(os.path.join(dn_temp, fn_temp), os.path.join(dn_final, fn_temp))
elif suffix == 'tgz':
    archive.extractall(dn_final)

# cleaning up the temporary folder
if suffix == 'tgz':
    os.remove(os.path.join(dn_temp, fn_temp))
os.rmdir(dn_temp)

if not db.populateFiles(scan_uuid):
    return resp.InternalServerError
```

Code 9: Second part of the ListOfScans.post method (in scan.py), storing the uploaded files and populating the database files table

Only now, that the files reside in a folder named with the scans UUID in the configured upload folder (*/uploads* by default), the actual scans can be started and the results returned to the client. This is handled by the third and final part of the `post` method, as shown in listing 10:

```
 1 issues = []
 2 if request.form.get('scanner'):
 3     if request.form['scanner'] == 'yara':
 4         scanners = [scanner.YARAScanner()]
 5     elif request.form['scanner'] == 'phpcs':
 6         scanners = [scanner.PHP_CodeSniffer()]
 7     else:
 8         return resp.InvalidScanner
 9 else:
10     scanners = [scanner.YARAScanner(), scanner.PHP_CodeSniffer()]
11 for s in scanners:
12     s.scan_folder(dn_final)
13     issues.extend(s.issues)
14     s.submit_to_db(scan_uuid)
15
16 files = db.get_file_count(scan_uuid)
17 db.update_scan(scan_uuid, stage='done', percentage=100, \
18                issuesFound=len(issues), files=files, analysed='now')
19 response = db.getScan(scan_uuid, request.headers.get('X-API-KEY'))
20 if request.form.get('withIssues') and request.form['withIssues'] == 'true':
21     response['issues'] = issues
22 return response
```

Code 10: Third part of the ListOfScans.post method (in scan.py), starting the analysis and storing and returning its results

Here we see another specificity of the prototypical character of this implementation: the lines 2 to 10 in the above listing assume that that there are only two specific scanner classes. If a new scanner class is added, this code also would have to be changed. In a productive implementation this should be refactored so that the only changes needed after a new scanner class is added is in the configuration file.

The actual scans are then started in the `for` loop in lines 11 to 14. This way we can either use only one or both (or in a productive setup many) scanners to analyse the uploaded files. All found issues are accumulated in the `issues` list, which are then returned with the response, if the client requested it.

Herein lies another prototype specificity: the API is designed in a way that it allows for asynchronous scans, that can be initiated by the POST to the `/scans` endpoint, and then retrieved independently by a GET to the `/scans/<string:id>` endpoint, tracking the progress of the actual scan. This should certainly be implemented in a scalable productive system, as scans might take longer depending on the number of available scanners, the submitted code bases and the request rates to the server. This prototype version nevertheless allows for a different agents/users to initiate the scan and then retrieve them asynchronously with the provided UUID. This would represent a workflow where a student submits their exercise code to some existing submission tool, which in turn submits the code to the SCRAP server and then returns, among other direct feedback, a link to the SCRAP scan with which they can investigate the found issues themselves.

Now, that we saw how the scan is initiated, let us take a look at the final and central ingredient: the scanner classes in the */resources/scanner.py* file.

Listing 11 displays the very succinct `ListOfScanners` resource class, which only returns the information about available scanners, when the `/scanners` endpoint of the API is requested. This is followed by the definition of the `Scanner` class, which is the parent class that every new scanner wrapper class has to be derived from:

```
from flask_restful import Resource
from flask import g, current_app
import db
import yara
import json
import os
import subprocess
from subprocess import PIPE

class ListOfScanners(Resource):
    def get(self):
        response = []
        for scanner in current_app.config['SCANNERS']:
            response.append(scanner.getMeta())
        return response

class Scanner():
    def __init__(self, name, slug=None, version=None, uri=None, comment=None):
        self.name = name
        self.slug = slug
        self.version = version
        self.uri = uri
        self.comment = comment
        self.issues = []

    def __repr__(self):
        return '<Scanner␣' + self.slug + '␣("' + self.name + '")'

    def getMeta(self):
        return {
            'name': self.name,
            'slug': self.slug,
            'version': self.version,
            'uri' : self.uri,
            'comment': self.comment,
        }

    def submit_to_db(self, uuid):
        db.submit_issues(uuid, self.issues)
```

Code 11: ListOfScanners and Scanner classes (in scanners.py)

The `Scanner` class does some initialisation of the information that is used in the `ListOfScanners` request handler. It also provides the `submit_issues` method, which only in turn calls the same named function in the `db` module. This in turn guarantees that the `post` method in the handler for the POST request on the list of scans, that was presented above, can access the list of issues that will has to be filled by the actual scanner wrapper classes, which are derived from this `Scanner` class. It also masks the access to the database, so the only thing a derived scanner wrapper class has to do, besides calling the actual scanner programs, is to fill the `self.issues` list with all found issues.

Now lets take a look at the first of the two available scanner wrappers classes, that facilitate the scan of an uploaded file or archive content with the corresponding scanner, which have to be set up accordingly. *PHP_CodeSniffer* was set up as described in the static analysers subsection 5.3.2 of the evaluation section 5.3, only that it was not set up in a users home directory but under the */opt/scrap/scanners* directory. Listing 12 shows the `PHP_CodeSniffer` wrapper class, that will facilitate it within the Flask web service:

```
class PHP_CodeSniffer(Scanner):
    def __init__(self):
        super().__init__(
            name='PHP_CodeSniffer',
            slug='phpcs',
            version='3.5.4',
            uri='https://github.com/squizlabs/PHP_CodeSniffer',
            comment='Using␣the␣[phpcs-security-audit␣
                v2](https://github.com/FloeDesignTechnologies/phpcs-security-audit)',
        )
        self.cli_pattern='php␣/opt/scrap/scanners/phpcs/phpcs.phar␣' +\
            '--standard=/opt/scrap/scanners/phpcs-sa/Security␣' +\
            '-s␣--report=json␣%s'

    def scan_folder(self, folder):
        self.issues = []
        p = subprocess.run([
            'php', '/opt/scrap/scanners/phpcs/phpcs.phar',
            '--standard=/opt/scrap/scanners/phpcs-sa/Security',
            '-s', '--report=json', folder
        ], stdout=PIPE, stderr=PIPE)
        findings = json.loads(p.stdout)

        for file, finding in findings['files'].items():
            path = file.rsplit(folder, 1)[1]
            if path[0] == '/':
                path = path[1:]
            for issue in finding['messages']:
                self.issues.append({
                    'source': {
                        'scanner': self.slug,
                        'rule': issue['source'],
                        'info': self.uri,
                        'cli': self.cli_pattern.replace('%s', path)
                    },
                    'type': issue['source'],
                    'explanation': issue['message'],
                    'affectedFiles': [
                        {
                            'path': path,
                            'lines': [
                                {
                                    'num': issue['line'],
                                    'characters': {
                                        'from': issue['column']
                                    },
                                    'text': '',
                                    'description': issue['type'] + '␣|␣' + \
                                        'severity:␣' + str(issue['severity'])
                                }
                            ]
                        }
                    ]
                })
        return self.issues
```

Code 12: PHP_CodeSniffer scanner wrapper class (in scanners.py)

In the initialisation, only the parents constructor method is called and the scanners meta information is set. Additionally a `cli_pattern` is set, which is used to provide information on how the scan could be done manually in a similar server setup.

The only other method that is provided by the wrapper is the `scan_foler` method, which starts a `php` subprocess that starts the actual `phpcs.phar` scanner and parses its JSON output into the `findings` dictionary. The rest of the method consists of two nested `for` loops which go through all findings and transform them into a dictionary as it is specified by the SCRAP API specification, and then added to the issues list.

While the transformation might seem cumbersome, that is all the magic a scanner wrapper class has to do. This is quite similar for the `YARAScanner`. As there is a Python binding for *yara*, that is installed with all the other requirements we only need to set up a */opt/scrap/scanners/yara/* directory, that contains the *pmf.yar* and *whitelist.yar* as well as the *pmf* and *whitelist* folders from *PHP Malware Finder*, that was described in the static scanner evaluation. Additionally we should put a *scrap.yar* file there with content from listing 13:

```
include "pmf.yar"

rule SQLi
{
  meta:
    issue = "Your code might be vulnerable to an SQL injection"
    reason = "The $id parameter seems to not be sanitized"
    info = "https://scrap/description/sqli"

  strings:
    $unsanitized = /\$id\s*=\s*\$_REQUEST\[\s*['"]id['"]\s*\]\s*;/ nocase
    $injection = /SELECT.*FROM.*WHERE.*=\s*'\$id'/ nocase

  condition:
    $unsanitized and $injection
}
```

Code 13: scrap.yar main rule file for the YARAScanner

This way we include the rule set that is already available from *PHP Malware Finder* and can extend it with our own rules. With this setup the `YARAScanner` class looks like in listing 14:

```
class YARAScanner(Scanner):
    def __init__(self):
        super().__init__(
            name='YARA',
            slug='yara',
            version='3.7.1',
            uri='https://virustotal.github.io/yara/',
            comment='Including␣the␣rule␣set␣of␣[PHP␣Malware␣
                Finder](https://github.com/jvoisin/php-malware-finder)',
        )
        self.cli_pattern='yara␣-r␣-w␣-s␣-m␣/opt/scrap/scanners/yara/scrap.yar␣%s'
        self.rules = yara.compile(filepath='/opt/scrap/scanners/yara/scrap.yar')

    def scan_folder(self, folder):
        self.issues = []
        for dir in os.walk(folder):
            for file in dir[2]:
                matches = self.rules.match(dir[0]+'/'+file)
                for m in matches:
                    issue = {
                        'source': {
                            'scanner': self.slug,
                            'rule': m.rule,
                            'info': self.uri,
                            'cli': self.cli_pattern.replace('%s', dir[0]+'/'+file, 1)
                        },
                        'type': m.rule,
                        'explanation': 'yara.' + m.rule,
                        'affectedFiles': [
                            {
                                'path': str(dir[0]+'/'+file)[len(folder)+1:],
                                'lines': [
                                ]
                            },
                        ]
                    }
                    description = ''
                    if m.meta:
                        if 'issue' in m.meta:
                            description += m.meta['issue'] + '␣|␣'
                        if 'reason' in m.meta:
                            description += m.meta['reason'] + '␣|␣'
                        if 'info' in m.meta:
                            description += 'More␣info␣at:␣' + m.meta['info']
                    for s in m.strings:
                        issue['affectedFiles'][0]['lines'].append({
                            'characters': {
                                'from': s[0],
                                'to': s[0] + len(s[2])
                            },
                            'text': str(s[2]),
                            'description': description,
                        })
                    self.issues.append(issue)
        return self.issues
```

Code 14: YARAScanner wrapper class (in scanners.py)

As we see, the concept is similar as with the `PHP_CodeSniffer` class, and can be reproduced for every other scanner wrapper class that we want to extend SCRAP to. With *yara* we only have to scan every file separately to transform the results to the format specified by the SCRAP API definition. This makes for more nestes `for` loops, but the principle stays the same.

The transformation done by those two scanner wrapper classes could also certainly improved beyond a prototype stage, by providing more information and a consistent offset scheme. For the purpose of a proof of concept the current implementation shows that there is not a lot of effort involved in attaching more scanners to the SCRAP platform.

5.2.4 Web UI

The core component of the SCRAP platform is the web service that was described in the last section. As will be highlighted in the SCRAP evaluation later, wen can fully use it from other tools and integrate it into other systems based on its OpenAPI specification. Yet, one of the crucial factors for the success of such a solution is how the resulting information is presented to the end users, that is, the introductory programming students who submitted their code to some other platform.

Therefore this project also contains a prototypical web user interface with which students can submit separate scans on their own, provided there is a either public account set up on the web service, or they are given an API key by their institution. Additionally the web UI can also be used as a front end for the display of issues and explanations, so that only minimal adaptations have to be set to integrate SCRAP into an institutions code submission system. The latter only has to provide the student with the link to the web UI, containing the UUID of the submitted scan and a corresponding API key, as privacy should also be respected for student's code submissions, despite my high regards for mutual and co-shared learning approaches (although in a productive implementation of SCRAP there could be a sharing and discussion feature that allows students to work through their issues collaboratively).

Due to the main importance of the web UI in terms of content presentation I will refrain from providing code listings here. All code for the client, including information how to set it up, can be found in the scrap-client repository on GitLab as well as its copy on the accompanying data disc.

In this section I want just provide a brief glance over the implementation approach. This was done with the progressive JavaScript framework Vue.js in version 2. For *Vue.js* there is als a Vue CLI, which fosters rapid prototyping by providing boilerplate code and standard tooling, including a development server, based on the popular webpack asset bundler. Then there is also the Vuex state management library and the and the Vue Router, which make it easy to build a consistent application state and access to the web service and to implement a URI scheme analogous to the web service. For the asynchronous API requests the promise based HTTP client axios was chosen and for a smooth and consistent user interface the BootstrapVue library was used, bringing the popular and responsive front-end component library Bootstrap 4 to *Vue.js* in native Vue components.

The current Web UI prototype provides features to upload and list scans, tied to an API key, and to inspect scans and all related issues. For every issue, if an according explanation was written, three tabs are shown with information on what the vulnerability, how to fix it and further references. This was modelled after the promising *SonarQube* dashboard and its display of *Security Hotspots*, which is evaluated in the following section as one of the 7 static analysis tools.

An evaluation of the SCRAP API and the web service as well as the web UI, follows in the next section, right after the evaluation of the scanners. Corresponding screenshots of the web UI can be found there.

5.3 Evaluation

The SCRAP prototype consists of three components that can be evaluated:

1. The (potential) scanners and their adaptation
2. The web service API and its implementation server
3. The web client

A major factor for the viability of the approach presented in this thesis is the availability of good F/LOSS static analysis tools to scan PHP code and then in turn generate useful feedback for SCRAP code submissions at the webservice. Therefore the analysis of the F/LOSS tools presented in subsection 2.2.1 in the *Platforms and Tools* section (2.2) of the literature review is of major interest.

The following subsection describes the vulnerability test data set that was used to evaluate the different tools. The subsection thereafter provides a detailed description of the individual tools. The last subsection in this section then evaluates the prototype implementations of the SCRAP web service and client. Additionally the evaluation of the scanners and the description of the test data set are also accessible in the scrap-scanner-eval repository on GitLab, and are linked to in the project website at https://scrap.tantemalkah.at.

5.3.1 Vulnerability test data

As a data set to test the scanners, a subset of the vulnerability files of the Damn Vulnerable Web Application (short: *DVWA*) by Ryan Dewhurst (Dewhurst Security, 2020) was chosen. Of particular interested are the SQLi and XSS vulnerabilities, because these are the most likely ones to be encountered in introductory web application programming courses. To be able to test single PHP files as well as folders containing a collection of PHP files and/or subfolders with PHP files, the single vulnerability files have been placed in the root of the vulnerability test data folder, and the DVWA folders for the different vulnerability types (SQLi, blind SQLI, reflected XSS, stored XSS and dom-based XSS) have been copied as a whole.

To consistently generate a folder with this test data set, the bash script vuln-data-copy.sh was created and can be used to reproduce this set from a freshly unpacked *DVWA* download. The script can be found in the scrap-scanner-eval repository, as well as in the attachment and the accompanying data disc's `scrap-scanner-eval` directory.

The choice to use a subset of the *DVWA* was made because it nicely features examples for common web application security vulnerabilities, and has especially good coverage of SQL injection and XSS vulnerabilites, which are the most likely and early ones to come up in introductory web application programming courses. An additional factor for this choice also was my previous and good experience with it. Beyond that it was also mentioned and used in literature found in my review (Sahu & Tomar, 2017).

While this set is a good and pragmatic choice for the context of this master thesis, future research should apply other approaches and data sets. Examples for such are presented by (Schuckert et al., 2017) and (Stivalet & Fong, 2016). Also a mixture of different categories of sources could be applied, as presented by (Sahu & Tomar, 2017), who use, among others, the *Damn Vulnerable Web Application* and *OWASP Mutillidae 2*.

5.3.2 Static analysers

As noted above, the available F/LOSS tools found through literature review and web searches is described in section 2.2.1. As the resulting list of tools is too long to evaluate every single tool in detail, and as many of those tools are not maintained anymore, out of the list of 19 tools only 7 have been chosen for a deeper evaluation. For all of them there is a separate folder in the scrap-scanner-eval repository and the corresponding folder on the accompanying data disc, containing a `notes.md` file, which describes:

1. The setup and usage of the tool
2. A general assesssment
3. Its viablity in regard to a test data set
4. Remarks on the adaptation potential for SCRAP

Most of those folders also include screenshots of the tool's general usage and its application to the test data. Some folders additionally contain small exemplary scripts or rules to highlight the potential for the tool's adaptation in SCRAP.

The following subsections describe the evaluation of each of the 7 tools in detail, followed by a general résumé. For a consistent setup scenario, that can be utilised in the SCRAP prototypes all tools have been placed in a *scanner* directory, so that the same setup can be put into a */opt/scrap/scanners* directory in a productive setup. All evaluations were done on Debian 10 minmal server system, that was sandboxed as a *kvm/libvirt* virtualised system on a Ubuntu 18.04 notebook. The *scanner* directory for the evaluation was put in the home directory of a non-privileged user account. This is the reference point for mentions of this directory in the following subsections.

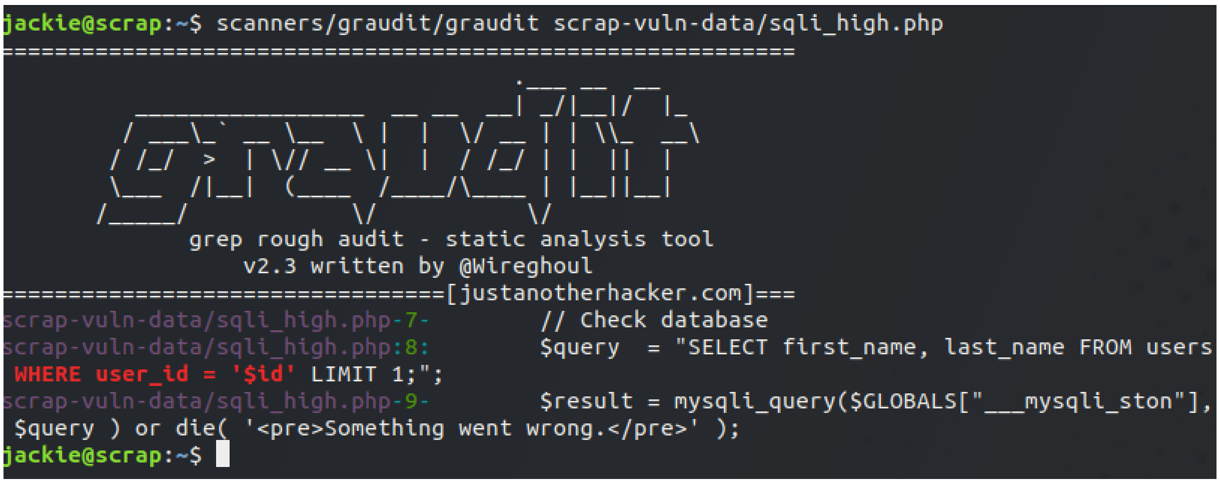
In the following sections not all screenshots that are available in the repository and on the accompanying data disc will be depicted, to save space for this paper (also) based medium. Nevertheless, for all mentioned screenshots - also those that are directly depicted here - the filename is provided and links to the picture file in the scrap-scanner-eval repository.

graudit

The **setup and usage** of this tool are quite easy, and well documented in its GitHub repository: https://github.com/wireghoul/graudit.

For non-permanent use, we just need to clone the repo and run `graudit` from the repo's root dir. The screenshot graudit01.png shows the cloned repo root content and a call of `graudit` without any arguments, providing a usage description of the tool.

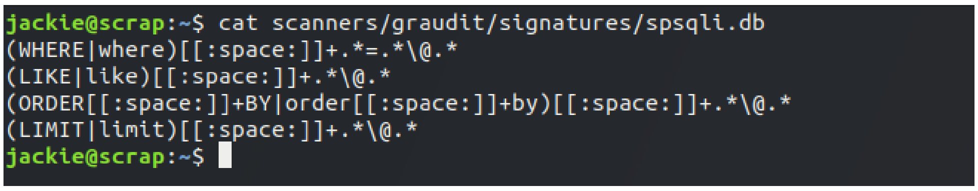
The easiest use is to call `graudit` just with one argument, the path with to a single file, as the graudit02.png hows, which is also depicted in figure 7. The same approach works not only for single files but also for a directory containing more files and directories.



```
jackie@scrap:~$ scanners/graudit/graudit scrap-vuln-data/sqli_high.php
===========================================================
                                      .___ __  __
          _________________  __ __  __| _/|__|/  |_
         / ___\_` __ \__  \ |  |  \/ __ | | \\_  __\
        / /_/  >  | \// __ \|  |  / /_/ | |  ||  |
        \___  /|__|  (____  /____/\____ | |__||__|
       /_____/            \/           \/
              grep rough audit - static analysis tool
                  v2.3 written by @Wireghoul
=================================[justanotherhacker.com]===
scrap-vuln-data/sqli_high.php-7-        // Check database
scrap-vuln-data/sqli_high.php:8:        $query  = "SELECT first_name, last_name FROM users
 WHERE user_id = '$id' LIMIT 1;";
scrap-vuln-data/sqli_high.php-9-        $result = mysqli_query($GLOBALS["___mysqli_ston"],
 $query ) or die( '<pre>Something went wrong.</pre>' );
jackie@scrap:~$
```

Figure 7: Simplest use of graudit on a single file

The signature/rule files are also easy to find and to clone/modify. All out-of-the-box rule-"databases" are found in the *signatures* folder, as the screenshot graudit03.png shows. The signature databases are just plain text files with one regular expression per line. Screenshot graudit04.png, depicted in figure 8 shows the example of potential SQL injections.



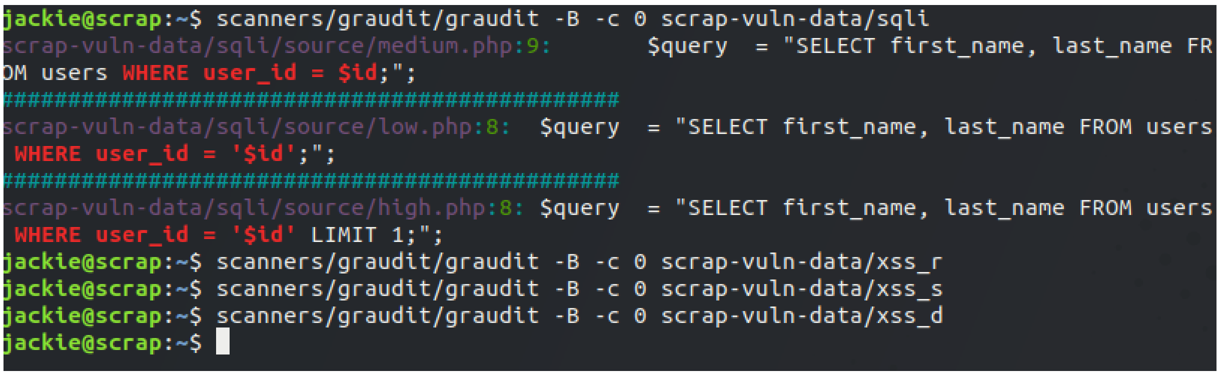
```
jackie@scrap:~$ cat scanners/graudit/signatures/spsqli.db
(WHERE|where)[[:space:]]+.*=.*\@.*
(LIKE|like)[[:space:]]+.*\@.*
(ORDER[[:space:]]+BY|order[[:space:]]+by)[[:space:]]+.*\@.*
(LIMIT|limit)[[:space:]]+.*\@.*
jackie@scrap:~$
```

Figure 8: The graudit rules for SQL injections

A **general assessment** of `graudit` is, that it is a simple regular-expression based grepper tool that seems to primarily support code auditors to find code segments which need careful analysis.

Applied to the test data it only found the SQL injection vulnerabilities. But it does not provide any information on what type of error/vulnerability/dodgy code it found and why, that is, based on which rules, as screenshot graudit05.png shows, also depicted in figure 9.

A full scan of the whole DVWA source generates more findings, but they are also quite unspecific. A quick and simple analysis with an out-of-the-box set up is not quite feasible for our purposes. But the tool can be easily extended with other rule databases. By using only specific databases on specific files provides the most value.



```
jackie@scrap:~$ scanners/graudit/graudit -B -c 0 scrap-vuln-data/sqli
scrap-vuln-data/sqli/source/medium.php:9:       $query  = "SELECT first_name, last_name FR
OM users WHERE user_id = $id;";
##############################################
scrap-vuln-data/sqli/source/low.php:8:  $query  = "SELECT first_name, last_name FROM users
 WHERE user_id = '$id';";
##############################################
scrap-vuln-data/sqli/source/high.php:8: $query  = "SELECT first_name, last_name FROM users
 WHERE user_id = '$id' LIMIT 1;";
jackie@scrap:~$ scanners/graudit/graudit -B -c 0 scrap-vuln-data/xss_r
jackie@scrap:~$ scanners/graudit/graudit -B -c 0 scrap-vuln-data/xss_s
jackie@scrap:~$ scanners/graudit/graudit -B -c 0 scrap-vuln-data/xss_d
jackie@scrap:~$ 
```

Figure 9: Results for graudit scans of 4 subfolders

For the **adaptation in SCRAP** a wrapper script would need to scan all individual files of the submission consecutively with each specific rule database. This way, for every code line found by `graudit`, the wrapper can produce additional information on which ruleset is responsible for this hit. In that way also information on specific SQLi, XSS and other vulnerability categories can be attached to the found code line.

phan

For the **setup and usage** of `phan` there are several options. For the use on a project basis, the installation through a composer dependency seems to be the most viable one. But for a quick evaluation, we can also just load the current *phan.phar* file from its release page, in this case version 2.5.0, and put it into our scanners collection under *scanners/phan*.

Phan needs *php-ast* in version 1.0.1 or higher as a dependency, but the Debian repository only provides it in version 0.1.6. The is easily solved with the three commands in listing 15.

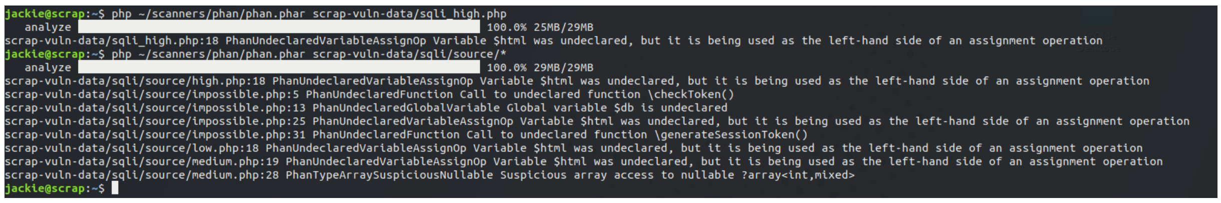
```
sudo apt install php-pear
sudo apt install php-dev
sudo pecl install ast-1.0.5
```

Code 15: Setting up php–ast for phan

After that we need to add `extension=ast.so` to the */etc/php/7.3/cli/php.ini*. Now phan can be run, listing available options by calling `php ~/scanners/phan/phan.phar --help`, as seen in the screenshot phan01.png.

It is also quite easy to run out-of-the box tests on single files or a directory's content as screenshot phan02.png shows, which is also depicted in figure 10.

For more complex setups there are many options to configure `phan` either via command line arguments or through a config file. The project documentation also suggests to start with strengthening ones analysis incrementally by starting with a relaxed analysis first and then increase the amount of things `phan` looks for in consecutive analysis runs (github.com/phan/phan, 2020).



```
jackie@scrap:~$ php ~/scanners/phan/phan.phar scrap-vuln-data/sqli_high.php
   analyze                                                             100.0% 25MB/29MB
scrap-vuln-data/sqli_high.php:18 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
jackie@scrap:~$ php ~/scanners/phan/phan.phar scrap-vuln-data/sqli/source/*
   analyze                                                             100.0% 29MB/29MB
scrap-vuln-data/sqli/source/high.php:18 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
scrap-vuln-data/sqli/source/impossible.php:5 PhanUndeclaredFunction Call to undeclared function \checkToken()
scrap-vuln-data/sqli/source/impossible.php:13 PhanUndeclaredGlobalVariable Global variable $db is undeclared
scrap-vuln-data/sqli/source/impossible.php:25 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
scrap-vuln-data/sqli/source/impossible.php:31 PhanUndeclaredFunction Call to undeclared function \generateSessionToken()
scrap-vuln-data/sqli/source/low.php:18 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
scrap-vuln-data/sqli/source/medium.php:19 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
scrap-vuln-data/sqli/source/medium.php:28 PhanTypeArraySuspiciousNullable Suspicious array access to nullable ?array<int,mixed>
jackie@scrap:~$
```

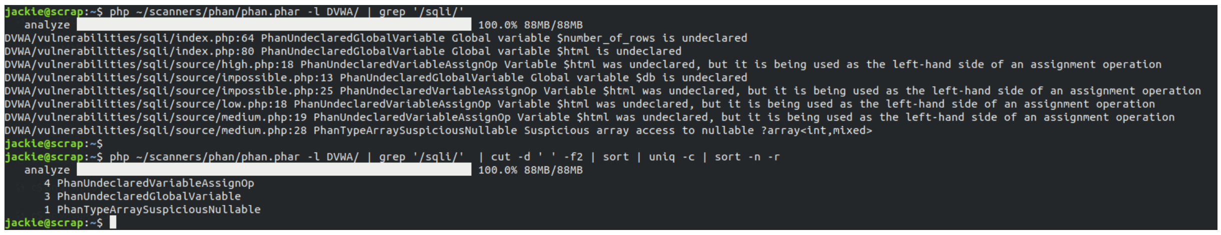
Figure 10: Simple call of phan with results

As a **general assessment** `phan` looks like a well developed static analyser for overall PHP code quality. It also has plugins/extensions for the editors *vim*, *emacs* and *VS Code*, which could make it quite useful for integration into introductory programming education. The plug-in/extension system is well documented and one could write security-specific plugins, although it is a lot less simple than with e.g. `graudit`, where you just have to add simple matching rules. In terms of security there is no specific focus.

Applied to the test data, none of the security issues were found. But it found a lot of general issues that are important for code quality, and which could not have been found by a simple grepper like `graudit`.

As an example, scanning the whole folder containing the SQLi related code templates of the *DVWA*, 25 of the 26 found issues are due to use of undeclared functions or variables, which is only the case because we scan a folder within the whole application, as show in screenshot phan03.png.

If we scan the whole *DVWA* folder and filter for only those issues coming from the *vulnerabilities/sqli/* folder, the hit count is reduced to 8, with 7 still coming from undeclared variables, as screenshot phan04.png shows, which is depicted in figure 11.



```
jackie@scrap:~$ php ~/scanners/phan/phan.phar -l DVWA/ | grep '/sqli/'
   analyze                                                             100.0% 88MB/88MB
DVWA/vulnerabilities/sqli/index.php:64 PhanUndeclaredGlobalVariable Global variable $number_of_rows is undeclared
DVWA/vulnerabilities/sqli/index.php:80 PhanUndeclaredGlobalVariable Global variable $html is undeclared
DVWA/vulnerabilities/sqli/source/high.php:18 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
DVWA/vulnerabilities/sqli/source/impossible.php:13 PhanUndeclaredGlobalVariable Global variable $db is undeclared
DVWA/vulnerabilities/sqli/source/impossible.php:25 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
DVWA/vulnerabilities/sqli/source/low.php:18 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
DVWA/vulnerabilities/sqli/source/medium.php:19 PhanUndeclaredVariableAssignOp Variable $html was undeclared, but it is being used as the left-hand side of an assignment operation
DVWA/vulnerabilities/sqli/source/medium.php:28 PhanTypeArraySuspiciousNullable Suspicious array access to nullable ?array<int,mixed>
jackie@scrap:~$
jackie@scrap:~$ php ~/scanners/phan/phan.phar -l DVWA/ | grep '/sqli/'  | cut -d ' ' -f2 | sort | uniq -c | sort -n -r
   analyze                                                             100.0% 88MB/88MB
      4 PhanUndeclaredVariableAssignOp
      3 PhanUndeclaredGlobalVariable
      1 PhanTypeArraySuspiciousNullable
jackie@scrap:~$
```

Figure 11: Scanning the DVWA SQLi vulnerability folder with phan

A definite advantage of `phan` compared to `graudit` is that it provides information about what it found. This helps to look for specific issues and provides an overall overview of the state of a whole application. Screenshot phan05.png for example is shows the first screen of a count of found issues for the whole *DVWA*, where the first three issue categories are *PhanUndeclaredProperty*, *PhanUndeclaredMethod* and *PhanUndeclaredClassPropert* with 170, 113 and 77 hits respectively.

In contrast to simple regex based grepping, `phan` does not just do simple pattern matching but works directly on the abstract syntax tree of the scanned PHP code. In that sense it is a true static code analyser, whereas `graudit` is just a pseudo analyser.

For the **adaptation in SCRAP** `phan` already provides categories and minimal explanations off the found issues. This can be used by a wrapper script. Unfortunately no security specific issues have been found in the test data. Therefore at least one plug-in would have to be developed for `phan` to focus on security issues.

PHP Malware Finder

The **setup and usage** for *PHP Malware Finder*, or short *PMF*, are quite simple. *PMF* works by applying YARA rules, so we need the `yara` package, which can be simply installed with `sudo apt install yara` on any Debian base system. Then we just clone the PMF repository and, to be consistent in our scanner usage, make a link called `pmf` to the *php-malware-finder* in the subdirectory, as shown in listing 16.

```
git clone https://github.com/jvoisin/php-malware-finder.git
ln -s php-malware-finder/php-malware-finder pmf
```

Code 16: Setting up PHP Malware Finder

Now we can scan our test data right away with the command in listing 17.

```
~/scanners/pmf/phpmalwarefinder ~/scrap-vuln-data
```

Code 17: Simple scan of whole test data set with PMF

As a **general assessment**, *PHP Malware Finder* is a good tool to scan a web host for deployed sites that use PHP code, in order to find malware like web shells and other dodgy or potentially obfuscated code. It relies on a pseudo-static analysis based on a set of YARA rules and is therefore rather easy to extend. Also the deployment is quite easy.

Applied to the test data, `pmf` did not find any of the vulnerabilites and it did find only minor other issues. Even applied to the whole vulnerable test data folder of the *DVWA*, only 4 general issues have been found, two coming from explanations with from links to a page on *pentestmonkey.net*, as screenshot pmf01.png shows.

To its defense it has to be mentioned, that its intention is not to strictly analyse code but to find known patterns for dodgy and potentially vulnerable code, and most of all to find known malware signatures in the analysed code base.

For the **adaptation in SCRAP**, not only a wrapper script would be needed but also a whole set of YARA rules have to be developed to find vulnerable code that is not detected by the provided YARA rule file in *PMF*.

A simple rule for exactly the issue in the *sqli_low.php* file of the vulnerable test data would be the one in listing 18.

```
rule SQLi
{
  strings:
    $unsanitized = /\$id\s*=\s*\$_REQUEST\[\s*['"]id['"]\s*\]\s*;/ nocase
    $injection = /SELECT.*FROM.*WHERE.*=\s*'\$id'/ nocase

  condition:
    $unsanitized and $injection
}
```

Code 18: A simple YARA rule to find the vulnerability in sqli_low.php

Put into our own *scrap.yar* file we can use the command in listing 19 to find the vulnerability with *PMF*:

```
~/scanners/pmf/phpmalwarefinder -v -c scrap.yar scrap-vuln-data/sqli_low.php
```

Code 19: PMF scan with custon YARA rule

But as *PMF* is basically just a wrapper for `yara`, bundled with a set of YARA rules, we can achieve an even better result (without the warning about unbounded `.*` regexp filters) by directly calling yara, which listing 20 shows:

```
yara -w -s scrap.yar scrap-vuln-data/sqli_low.php
```

Code 20: Calling yara directly with custom rule

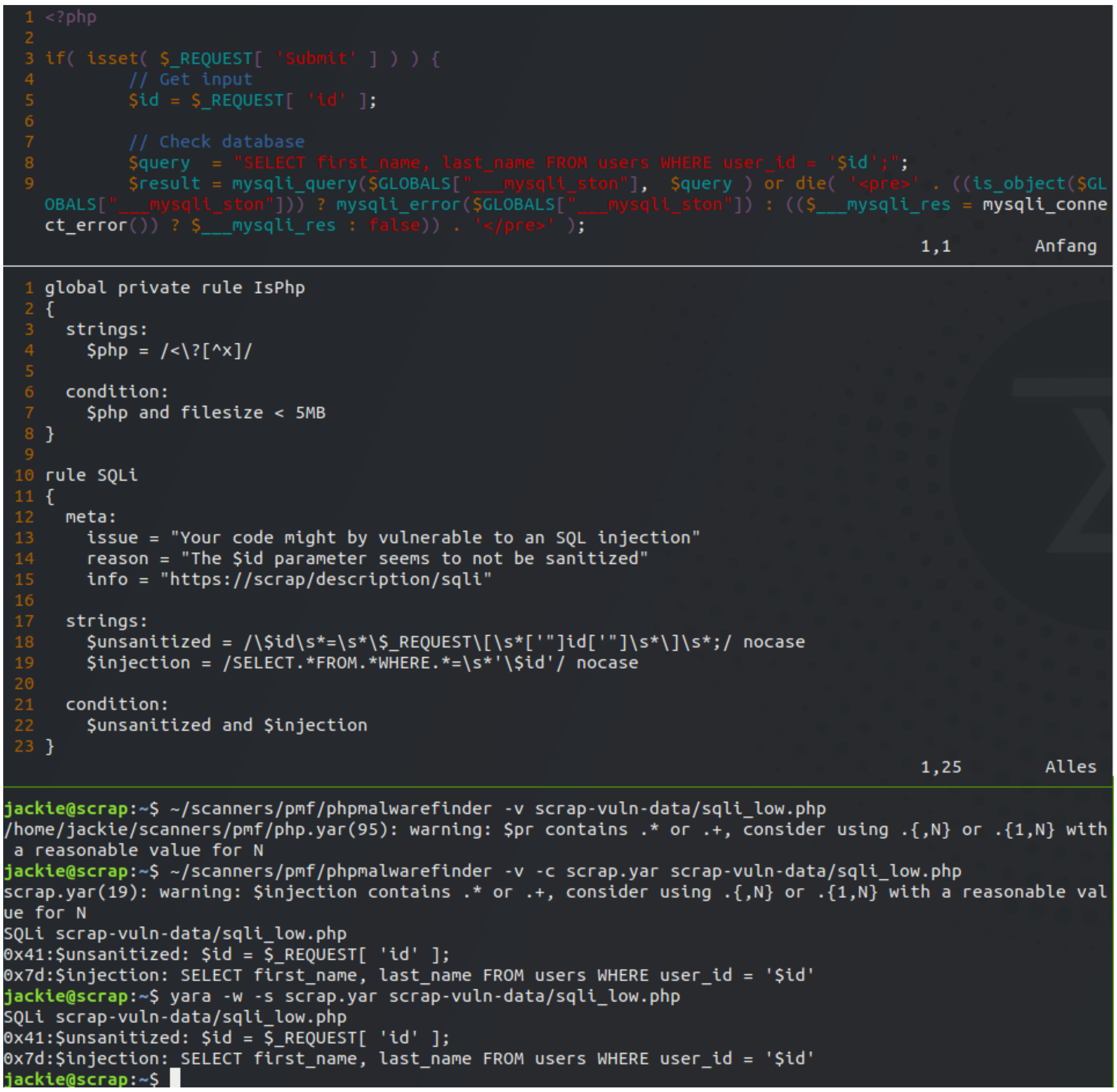
This is demonstrated in screenshot pmf02.png, which is also depicted in figure 12. In the screenshot we see a tiled bash view (using `tmux`), with the uppermost window showing the vulnerable PHP code that gets analysed, the middel part showing our own YARA rule file, and the bottom window showing the output of *PMF* both without and with our own rule as well as the output of calling `yara` directly using our own rule.

The direct use of `yara` also entails the advantage that we can add meta information to our rule, which then can be output by adding the `-m` option to the call of `yara` as it was depicted in figure 12. The annotated YARA rule then looks as in listing 21.

```
rule SQLi
{
  meta:
    issue = "Your code might by vulnerable to an SQL injection"
    reason = "The $id parameter seems to not be sanitized"
    info = "https://scrap/description/sqli"

  strings:
    $unsanitized = /\$id\s*=\s*\$_REQUEST\[\s*['"]id['"]\s*\]\s*;/ nocase
    $injection = /SELECT.*FROM.*WHERE.*=\s*'\$id'/ nocase

  condition:
    $unsanitized and $injection
}
```

Code 21: Annotated YARA rule to find the vulnerability in sqli_low.php

The screenshot pmf03.png, depicted in figure 13 shows the annotated output of `yara` with our basic rule for finding a specific SQL injection.

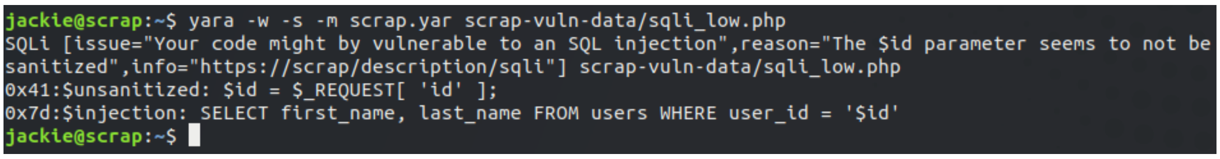


```
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
        // Get input
        $id = $_REQUEST[ 'id' ];

        // Check database
        $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die( '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res = mysqli_connect_error()) ? $___mysqli_res : false)) . '</pre>' );
                                                                                  1,1         Anfang
```

```
global private rule IsPhp
{
  strings:
    $php = /<\?[^x]/

  condition:
    $php and filesize < 5MB
}

rule SQLi
{
  meta:
    issue = "Your code might by vulnerable to an SQL injection"
    reason = "The $id parameter seems to not be sanitized"
    info = "https://scrap/description/sqli"

  strings:
    $unsanitized = /\$id\s*=\s*\$_REQUEST\[\s*['"]id['"]\s*\]\s*;/ nocase
    $injection = /SELECT.*FROM.*WHERE.*=\s*'\$id'/ nocase

  condition:
    $unsanitized and $injection
}
                                                                                  1,25         Alles
```

```
jackie@scrap:~$ ~/scanners/pmf/phpmalwarefinder -v scrap-vuln-data/sqli_low.php
/home/jackie/scanners/pmf/php.yar(95): warning: $pr contains .* or .+, consider using .{,N} or .{1,N} with
 a reasonable value for N
jackie@scrap:~$ ~/scanners/pmf/phpmalwarefinder -v -c scrap.yar scrap-vuln-data/sqli_low.php
scrap.yar(19): warning: $injection contains .* or .+, consider using .{,N} or .{1,N} with a reasonable val
ue for N
SQLi scrap-vuln-data/sqli_low.php
0x41:$unsanitized: $id = $_REQUEST[ 'id' ];
0x7d:$injection: SELECT first_name, last_name FROM users WHERE user_id = '$id'
jackie@scrap:~$ yara -w -s scrap.yar scrap-vuln-data/sqli_low.php
SQLi scrap-vuln-data/sqli_low.php
0x41:$unsanitized: $id = $_REQUEST[ 'id' ];
0x7d:$injection: SELECT first_name, last_name FROM users WHERE user_id = '$id'
jackie@scrap:~$
```

Figure 12: Using yara to do PMF's work with our own rule



```
jackie@scrap:~$ yara -w -s -m scrap.yar scrap-vuln-data/sqli_low.php
SQLi [issue="Your code might by vulnerable to an SQL injection",reason="The $id parameter seems to not be
sanitized",info="https://scrap/description/sqli"] scrap-vuln-data/sqli_low.php
0x41:$unsanitized: $id = $_REQUEST[ 'id' ];
0x7d:$injection: SELECT first_name, last_name FROM users WHERE user_id = '$id'
jackie@scrap:~$
```

Figure 13: Output of yara, when using our annotated rule on the sqli_low.php file

This all shows that it does not make sense to adapt *PMF* directly. If anything, we could potentially use some rules out of its YARA rule set and extend it with our own YARA rules. This then would be a viable option, when multiple scanners are used, because this approach makes it rather easy to add new rules for known issues.

PHPMD

For **setup and usage** *PHPMD* can be used directly from a *.phar* package released by the project. It comes with different sets of rules and one has to choose which ruleset to use for the analysis, while a mixture of different sets and custom ruleset files can be chosen.

To make it usable in our scanner test array, we just download the *phar* package into a new *phpmd* subfolder as done in listing 22:

```
mkdir ~/scanners/phpmd && cd ~/scanners/phpmd
wget https://phpmd.org/static/latest/phpmd.phar
```

Code 22: The simple setup of PHPMD

In its simplest form we can call it like in listing 23, using only the *cleancode* ruleset and printing the results as a *json* object:

```
php ~/scanners/phpmd/phpmd.phar ~/scrap-vuln-data json cleancode
```

Code 23: Simple scan of whole test data set with PHPMD

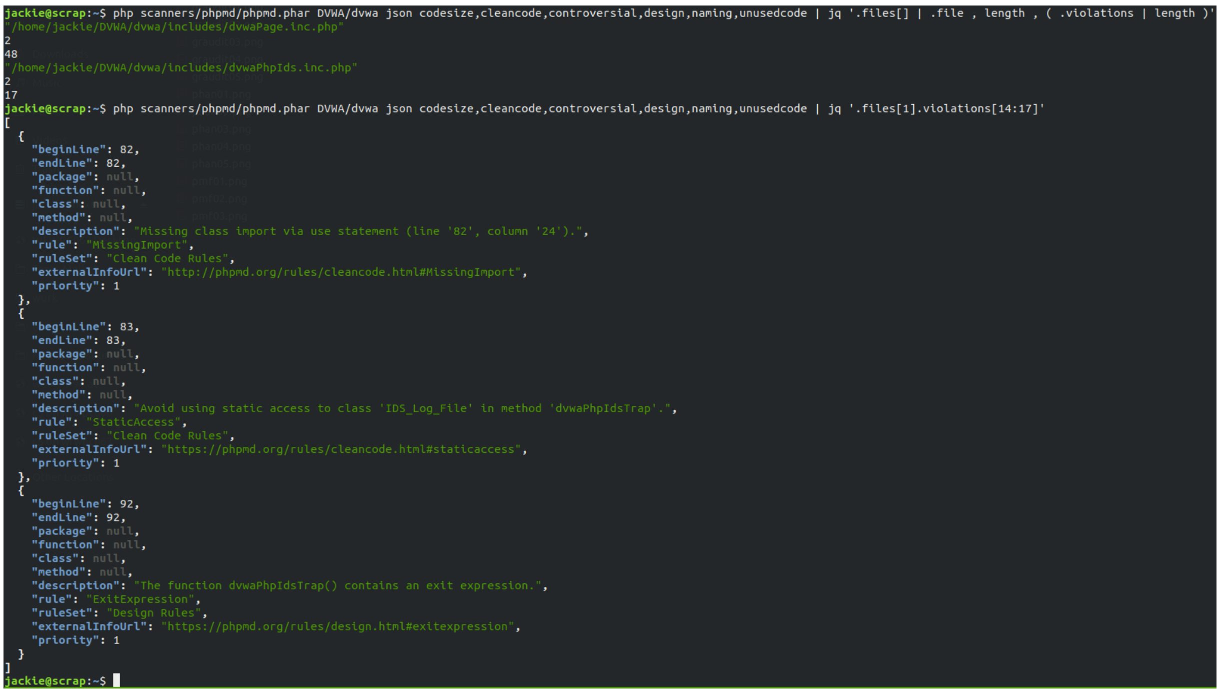
As a **general assessment** *PHPMD* is a good general purpose tool for static analysis of PHP code bases, when it comes to improving code quality. It comes with several rule sets integrated and has good documentation for how to use them and how to create custom rule sets based on the existing detection functions. It does not seem to have any specific focus on code security more than making sure that code is functional, readable and bug-free (which is in itself already an important aspect of good coding practice).

Applied to the test data out-of-the-box, *PHPMD* does not provide any findings, as screenshot phpmd01.png shows, one first with a *text* output formant and then with a *json* output format. To make sure that this is not due to a misconfiguration, I also scanned the whole *DVWA* code base with it and got a lot of hits.

Screenshot phpmd02.png is depicted in figure 14 and shows an example search running `phpmd.phar` on the *DVWA/dvwa* folder, filtering the `json` results, first to check which files have triggered rules and how many, and secondly outputting only the last three findings from the second file in the result.

This shows the very handy feature of providing the output in JSON format, as well as the good annotation of the rules and their categories, as well as where it was found.

For the **adaptation in SCRAP**, *PHPMD* looks like a solid framework that could be used for its versatile output handling. Custom rule sets could be written in order to catch vulnerable code, and the annotation feature of the rules would fit well into a SCRAP toolchain that should finally provide appropriate descriptions of the findings to the user.



```
jackie@scrap:~$ php scanners/phpmd/phpmd.phar DVWA/dvwa json codesize,cleancode,controversial,design,naming,unusedcode | jq '.files[] | .file , length , ( .violations | length )'
"/home/jackie/DVWA/dvwa/includes/dvwaPage.inc.php"
2
48
"/home/jackie/DVWA/dvwa/includes/dvwaPhpIds.inc.php"
2
17
jackie@scrap:~$ php scanners/phpmd/phpmd.phar DVWA/dvwa json codesize,cleancode,controversial,design,naming,unusedcode | jq '.files[1].violations[14:17]'
[
  {
    "beginLine": 82,
    "endLine": 82,
    "package": null,
    "function": null,
    "class": null,
    "method": null,
    "description": "Missing class import via use statement (line '82', column '24').",
    "rule": "MissingImport",
    "ruleSet": "Clean Code Rules",
    "externalInfoUrl": "http://phpmd.org/rules/cleancode.html#MissingImport",
    "priority": 1
  },
  {
    "beginLine": 83,
    "endLine": 83,
    "package": null,
    "function": null,
    "class": null,
    "method": null,
    "description": "Avoid using static access to class 'IDS_Log_File' in method 'dvwaPhpIdsTrap'.",
    "rule": "StaticAccess",
    "ruleSet": "Clean Code Rules",
    "externalInfoUrl": "https://phpmd.org/rules/cleancode.html#staticaccess",
    "priority": 1
  },
  {
    "beginLine": 92,
    "endLine": 92,
    "package": null,
    "function": null,
    "class": null,
    "method": null,
    "description": "The function dvwaPhpIdsTrap() contains an exit expression.",
    "rule": "ExitExpression",
    "ruleSet": "Design Rules",
    "externalInfoUrl": "https://phpmd.org/rules/design.html#exitexpression",
    "priority": 1
  }
]
jackie@scrap:~$
```

Figure 14: Output of PHPMD filtered through json, when run on the DVWA/dvwa folder

Nevertheless, one would have to also develop additional detector functions that could be used in those rule sets, as it seems that not many vulnerabilities could be described based on the existing rules. The decision whether to go down that road depends on whether a scanner with better out-of-the-box results can be found and adopted for SCRAP.

PHPStan

For **setup and usage** of *PHPStan* the PHP dependency manager *composer* seems to be the most straight-forward option, as we do not have to track all dependencies and *composer* itself is packaged in the Debian repository. This makes the setup as brief as in listing 24

```
sudo apt install composer
mkdir ~/scanners/phpstan && cd ~/scanners/phpstan
composer require --dev phpstan/phpstan
ln -s vendor/phpstan/phpstan/phpstan
```

Code 24: Setup of PHPStan

A first execution without any arguments already provides a a well defined user interface, as seen in screenshots phpstan01.png and phpstan02.png.

In our setup a single file or a directory can be scanned with the commands ins listing 25:

```
~/scanners/phpstan/phpstan analyse ~/scrap-vuln-data/sqli_low.php
~/scanners/phpstan/phpstan analyse ~/scrap-vuln-data/sqli/
```

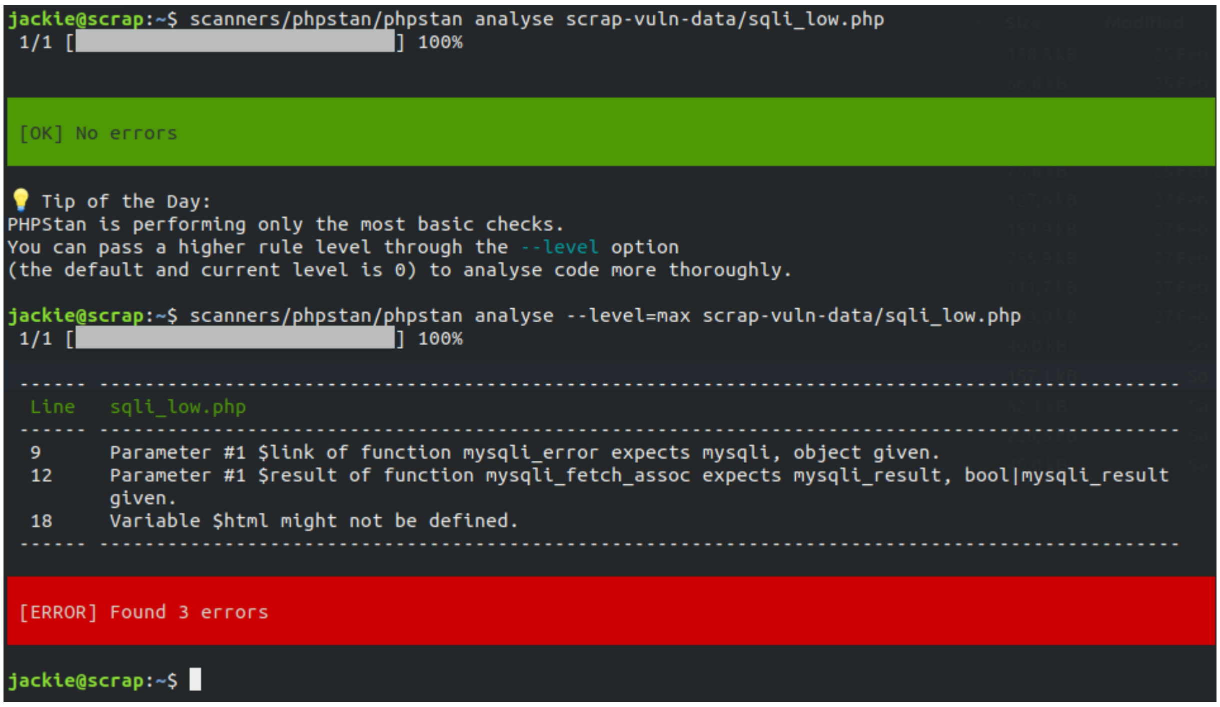
Code 25: Scanning a file and directory with PHPStan

While a lot of configuration and customization of scans can be applied through a *php-stan.neon* file, one important option that can be set directly from the command line is the rule level (currently between 0 for loose and 9 for strict). So if we want to catch most of the issues we would apply a level 9 or `max` to the search, as in listing 26:

```
~/scanners/phpstan/phpstan analyse --level=max ~/scrap-vuln-data/sqli_low.php
~/scanners/phpstan/phpstan analyse --level=max ~/scrap-vuln-data/sqli/
```

Code 26: Scanning a file and directory with PHPStan with the maximum analysis level

A comparison of the single file scan for *sql_low.php* with the default level 0 and the max level, can be seen in screenshot phpstan03.png, also depicted in figure 15.



```
jackie@scrap:~$ scanners/phpstan/phpstan analyse scrap-vuln-data/sqli_low.php
 1/1 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%

 [OK] No errors

💡 Tip of the Day:
PHPStan is performing only the most basic checks.
You can pass a higher rule level through the --level option
(the default and current level is 0) to analyse code more thoroughly.

jackie@scrap:~$ scanners/phpstan/phpstan analyse --level=max scrap-vuln-data/sqli_low.php
 1/1 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%

 ------ ---------------------------------------------------------------------------------
  Line   sqli_low.php
 ------ ---------------------------------------------------------------------------------
  9      Parameter #1 $link of function mysqli_error expects mysqli, object given.
  12     Parameter #1 $result of function mysqli_fetch_assoc expects mysqli_result, bool|mysqli_result
         given.
  18     Variable $html might not be defined.
 ------ ---------------------------------------------------------------------------------

 [ERROR] Found 3 errors

jackie@scrap:~$ █
```

Figure 15: Two PHPStan scans of a single file with different analysis levels

As a **general assessment** it can be stated that the main aim of *PHPStan* was to reduce the necessity of writing tests, as its author wrote in 2016 in a blog post on medium.com (Mirtes, 2016).

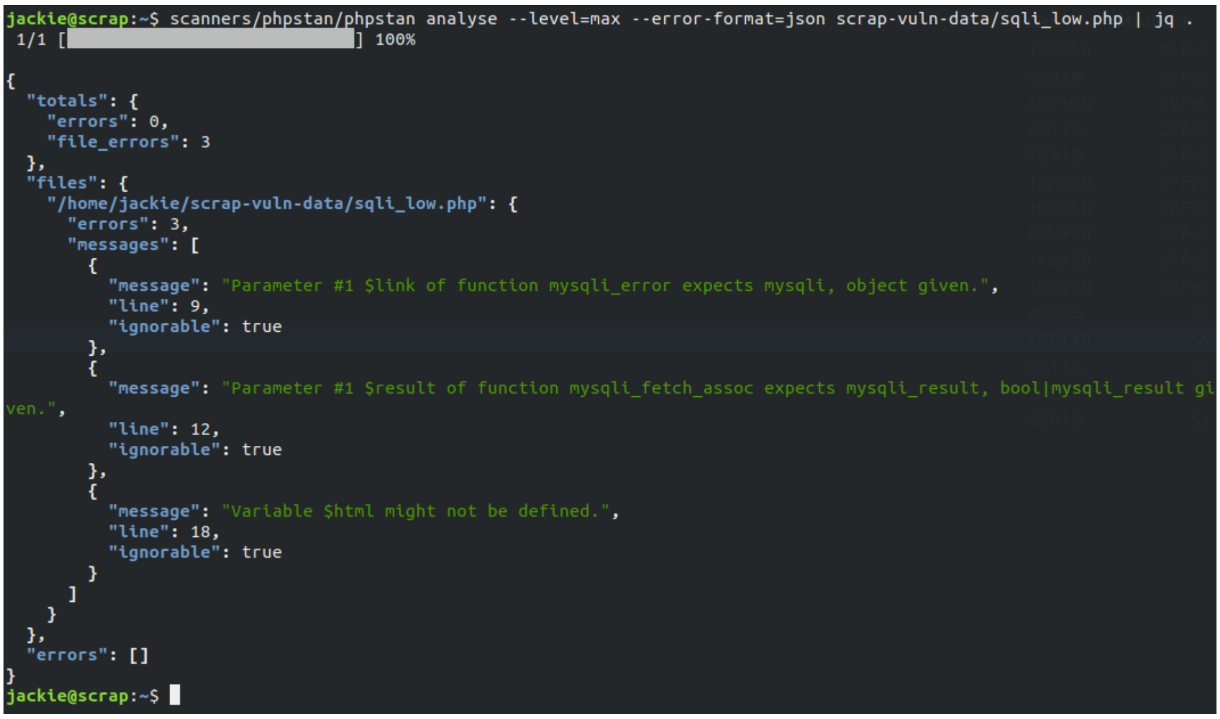
When we look at the initial checks, it becomes clear that *PHPStan* was made for larger PHP projects applying current web application software engineering paradigms and PHP practices, like e.g. using strict typing features introduced in PHP 7. *PHPStan* seems to be under constant development, and since the mentioned blog post, a lot has happened. This makes it a promising framework for static analysis of PHP code bases, although it aims at general purpose quality of code analysis and does not have an explicit focus on secure code. But the development process seems solid and very transparent (see the author's other medium.com blog posts), which would provide a good opportunity to collaborate and extend the project with specific vulnerability analyses.

PHPStan already has several framework-specific extensions, e.g. for *Doctrine*, *PHPUnit*, *Symfony*, and a lot more. Additionally there are unofficial extensions for e.g. *Laravel*, *Drupal*, *WordPress*, *TYPO3*, and many more. The project also has its own online playground at phpstan.org, which helps new and interested users to test out things first.

Aside from the technical aspects of the project, it is also very interesting to look at an April 2018 post (Mirtes, 2018*a*) and a December 2018 post (Mirtes, 2018*b*) from *PHPStan's* author on medium.com, outlining the problem with monetization and the available time to develop the project as F/LOSS. This points to an issue that probably many of the other projects featured in the literature review and the listing of F/LOSS PHP static analysers in section 2.2.1 had, and it might be also the reason why there are a lot of security-focused scanners out there, but many of them are not maintained any more.

When **applied to the test data** all tests have been run with the `--level=max` option, which produced around 50% more errors than with the default level of 0. Nevertheless no single vulnerability was found and most of the errors originated from references to *DVWA* functions and global variables not defined in the vulnerability test data.

For the **adaptation in SCRAP**, from a framework point of view, *PHPStan* could be a viable option, if security analysis can be integrated at some point. As already highlighted above, the CLI is well-designed, and also *json* and other output options are available, as seen in screenshot phpstan04.png, which is also depicted in figure 16.



```
jackie@scrap:~$ scanners/phpstan/phpstan analyse --level=max --error-format=json scrap-vuln-data/sqli_low.php | jq .
 1/1 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%

{
  "totals": {
    "errors": 0,
    "file_errors": 3
  },
  "files": {
    "/home/jackie/scrap-vuln-data/sqli_low.php": {
      "errors": 3,
      "messages": [
        {
          "message": "Parameter #1 $link of function mysqli_error expects mysqli, object given.",
          "line": 9,
          "ignorable": true
        },
        {
          "message": "Parameter #1 $result of function mysqli_fetch_assoc expects mysqli_result, bool|mysqli_result gi
ven.",
          "line": 12,
          "ignorable": true
        },
        {
          "message": "Variable $html might not be defined.",
          "line": 18,
          "ignorable": true
        }
      ]
    }
  },
  "errors": []
}
jackie@scrap:~$ █
```

Figure 16: PHPStan results in JSON notation

The online playground would provide SCRAP users an already available platform to test their code pieces individually outside SCRAP, and adopt a quality- and security-based coding practice for their future work.

PHP_CodeSniffer

For **setup and usage** of *PHP_CodeSniffer*, we can also use *composer*, which was already installed for the setup of *PHPStan*. Additionally, *PHP_CodeSniffer* is also released as phar files and therefore quite easy to set up for an evaluative usage, as shown in listing 27:

```
mkdir ~/scanners/phpcs && cd ~/scanners/phpcs
wget https://squizlabs.github.io/PHP_CodeSniffer/phpcs.phar
```

Code 27: Setup of PHP_CodeSniffer

Once set up this way, we can use the commands in listing 28 to get some usage information and scan a single file with out-of-the-box settings:

```
php ~/scanners/phpcs/phpcs.phar -h
php ~/scanners/phpcs/phpcs.phar ~/scrap-vuln-data/sqli_low.php
```

Code 28: Simple usage of PHP_CodeSniffer

Screenshot phpcs02.png shows the standard output, which is based on applying the PEAR Coding Standard.

With phpcs-security-audit v2 there is a set of security-specific rules for *PHP_CodeSniffer* available, so we want to do our tests on this basis. Therefore we clone the *phpcs-security-audit v2* repo to our scanners directory, and create a shorter link for brevity, as shown in listing 29:

```
cd ~/scanners
git clone https://github.com/FloeDesignTechnologies/phpcs-security-audit.git
ln -s phpcs-security-audit phpcs-sa
```

Code 29: Setup of the phpcs–security–audit v2

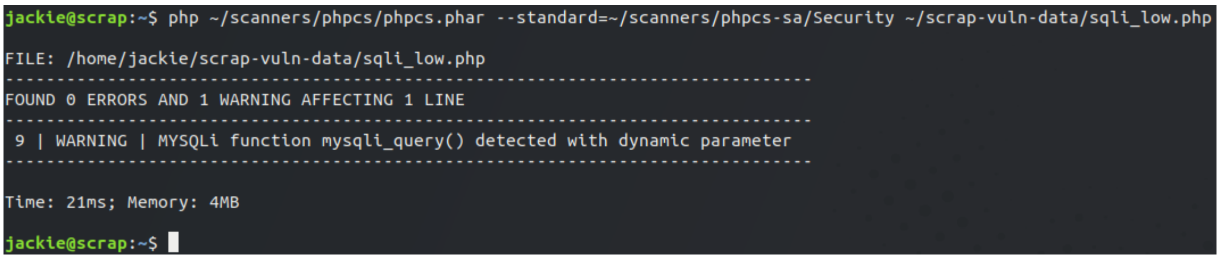
Now we can use its *Security* standard for the same scan as above, modifying the command as in listing 30:

```
php ~/scanners/phpcs/phpcs.phar --standard=~/scanners/phpcs-sa/Security
    ~/scrap-vuln-data/sqli_low.php
```

Code 30: Using PHP_CodeSniffer with the phpcs–security–audit v2 standard

The results of the above command are shown in screenshot phpcs03.png and depicted in figure 17).

As a **general assessment** *PHP_CodeSniffer* itself is a well developed and mature framework for general purpose code quality analysis, comparable to *PHPStan*, but with even more configurability and different coding standards as analytical reference to choose from, all on board. Interestingly, both projects have 179 contributors listed on GitHub (checked on 2020-03-05), but while *PHPStan* started in 2015, *PHP_CodeSniffer* is already around since 2006.



```
jackie@scrap:~$ php ~/scanners/phpcs/phpcs.phar --standard=~/scanners/phpcs-sa/Security ~/scrap-vuln-data/sqli_low.php

FILE: /home/jackie/scrap-vuln-data/sqli_low.php
--------------------------------------------------------------------------------
FOUND 0 ERRORS AND 1 WARNING AFFECTING 1 LINE
--------------------------------------------------------------------------------
 9 | WARNING | MYSQLi function mysqli_query() detected with dynamic parameter
--------------------------------------------------------------------------------

Time: 21ms; Memory: 4MB

jackie@scrap:~$ █
```

Figure 17: Result of a simple PHP_CodeSniffer scan with the phpcs-security-audit v2 standard

The maturity of the project is also visible in good documentation, a clear and transparent version numbering and release scheme and good extendability, especially when it comes to creating own coding standards.

Besides its use for static analysis it also comes with an additional script for automatic correction of coding standard violations. The best general outline on what it is and its intent can maybe taken directly from its Documentation page:

> “PHP_CodeSniffer is a set of two PHP scripts; the main phpcs script that tokenizes PHP, JavaScript and CSS files to detect violations of a defined coding standard, and a second phpcbf script to automatically correct coding standard violations. PHP_CodeSniffer is an essential development tool that ensures your code remains clean and consistent.
>
> A coding standard in PHP_CodeSniffer is a collection of sniff files. Each sniff file checks one part of the coding standard only. Multiple coding standards can be used within PHP_CodeSniffer so that the one installation can be used across multiple projects. The default coding standard used by PHP_CodeSniffer is the PEAR coding standard.” (github.com/squizlabs/PHP_CodeSniffer, 2016)

Applied to the test data, when we run *PHP_CodeSniffer* out-of-the-box (without the *phpcs-security-audit v2*) rule set, the results are similarly generic like with *PHPStan* or the other general purpose static analysers. Yet, the structure of the rule sets as “Coding Standards” and the availability of different on-board standards gives it a slightly better feeling for how to customize and how to trace and explain bad code.

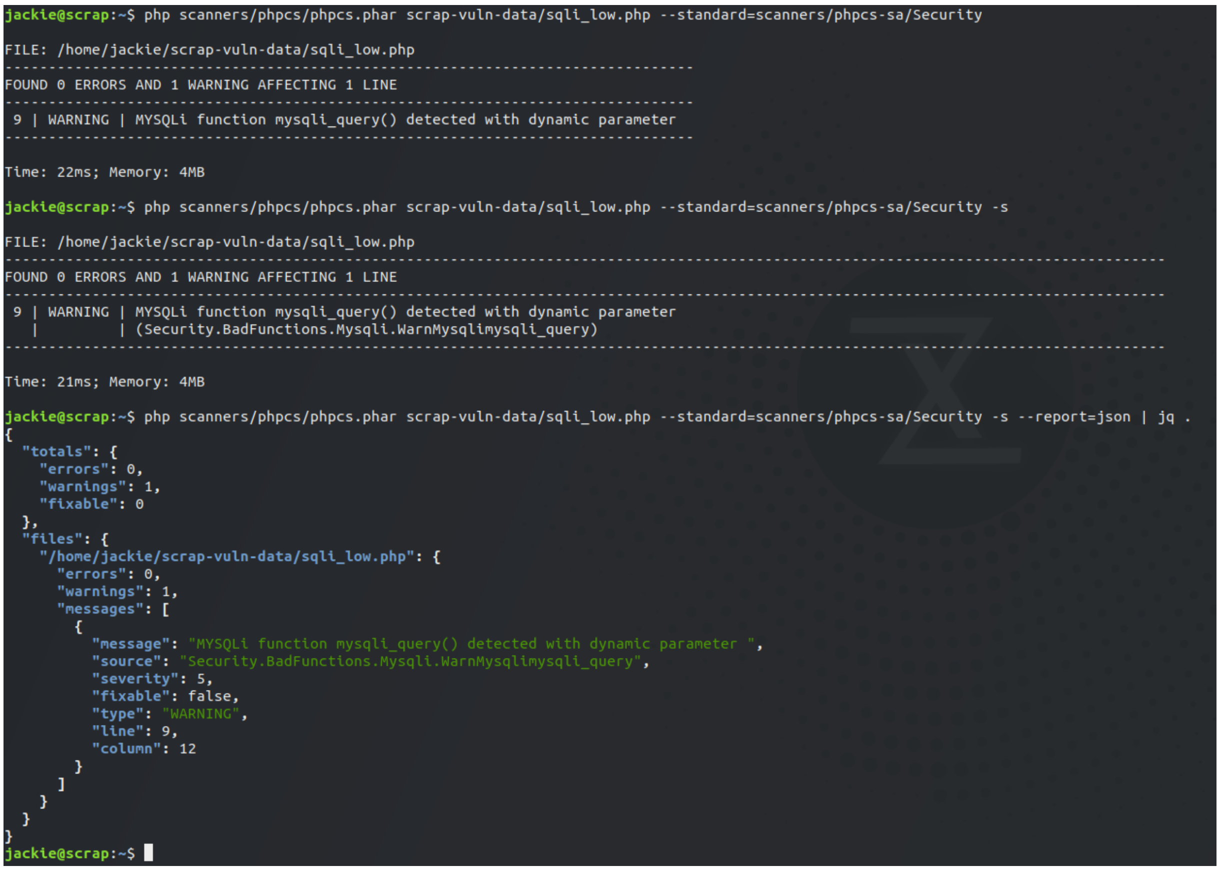
But when we apply the *phpcs-security-audit v2* coding standard on our test data we already get a good coverage of our vulnerabilities, without too many false positives or issues relating to other coding standards.

The tool does not tell which specific vulnerability is found, but points to the exact location where the vulnerability hits the application, as seen in screenshot phpcs04.png, where we compare the analysis for an SQL injection and a stored XSS vulnerability. Both resulted in the same issue found, which is acutally quite accurate, because in many cases stored XSS attacks build on available SQL injection vulnerabilities. Nevertheless, the rule set can certainly be improved to spot code pieces where specifically a XSS attack could happen due to unsanitized script output.

For the **adaptation in SCRAP** *PHP_CodeSniffer* with the *phpcs-security-audit v2* rule set seems to be the most promising candidate for adaptation in SCRAP. Also there is good documentation on how to write ones own coding standards, including the XML rule sets and the PHP code for the individual Sniffs used in those rule sets.

There are also a lot of options described on the Advanced Usage section of the documentation, which can be used to limit analysis to specific sniffs and to filter warnings and errors based on severity. And there is also a well documented and broad range of Reporting options.

Additionally, we might not only get information about the specific sniffs that found a vulnerability, but also output this in JSON format, as screenshot phpcs05.png shows, which is depicted in figure 18.



```
jackie@scrap:~$ php scanners/phpcs/phpcs.phar scrap-vuln-data/sqli_low.php --standard=scanners/phpcs-sa/Security

FILE: /home/jackie/scrap-vuln-data/sqli_low.php
-------------------------------------------------------------------------------
FOUND 0 ERRORS AND 1 WARNING AFFECTING 1 LINE
-------------------------------------------------------------------------------
 9 | WARNING | MYSQLi function mysqli_query() detected with dynamic parameter
-------------------------------------------------------------------------------

Time: 22ms; Memory: 4MB

jackie@scrap:~$ php scanners/phpcs/phpcs.phar scrap-vuln-data/sqli_low.php --standard=scanners/phpcs-sa/Security -s

FILE: /home/jackie/scrap-vuln-data/sqli_low.php
-------------------------------------------------------------------------------------------------------------------------------------
FOUND 0 ERRORS AND 1 WARNING AFFECTING 1 LINE
-------------------------------------------------------------------------------------------------------------------------------------
 9 | WARNING | MYSQLi function mysqli_query() detected with dynamic parameter
   |         | (Security.BadFunctions.Mysqli.WarnMysqlimysqli_query)
-------------------------------------------------------------------------------------------------------------------------------------

Time: 21ms; Memory: 4MB

jackie@scrap:~$ php scanners/phpcs/phpcs.phar scrap-vuln-data/sqli_low.php --standard=scanners/phpcs-sa/Security -s --report=json | jq .
{
  "totals": {
    "errors": 0,
    "warnings": 1,
    "fixable": 0
  },
  "files": {
    "/home/jackie/scrap-vuln-data/sqli_low.php": {
      "errors": 0,
      "warnings": 1,
      "messages": [
        {
          "message": "MYSQLi function mysqli_query() detected with dynamic parameter ",
          "source": "Security.BadFunctions.Mysqli.WarnMysqlimysqli_query",
          "severity": 5,
          "fixable": false,
          "type": "WARNING",
          "line": 9,
          "column": 12
        }
      ]
    }
  }
}
jackie@scrap:~$
```

Figure 18: Differnt output options of PHP_CodeSniffer

SonarQube

The **setup and usage** of *SonarQube* is trivial as in all the other scanners tested so far, as it is not a single tool but its own platform facilitating a range of tools and interfaces.

The SonarQube Server can be installed manually from a ZIP archive, but there is also a docker container available, which is especially helpful for evaluative purposes. For a productive setup the manual installation might make more sense, depending on available hardware resources.

After installing docker as described in the Get Docker Engine - Community for Debian article of the Docker documentation (docker docs, 2020), the SonarQube server can be started with the following a one-liner as in listing 31:

```
sudo docker run -d --name sonarqube -p 9000:9000 sonarqube
```

Code 31: Starting the SonarQube server with docker

After the start of the server the web interface is available on http://localhost:9000, or http://scrap:9000, if the hostname is configured as in our test setup. The credentials for an initial login can be found on the Get Started in Two Minutes Guide of the SonarQube documentation (SonarQube, 2020).

To scan a code base we first have to create a project in SonarQube, where we have to choose an API token, the language to analyse and the operating system. We then get a download link for the scanner and some documentation on how to start a scan, as seen in screenshot sonarqube01.png.

The scanner can be installed on any host that can communicate with the SonarQube server. In our case it is the same host. While there is also a *sonar-scanner-cli* Docker image, to be consistent in the test setup, I installed the scanner manually in the */scanners* directory with the commands in listing 32:

```
cd ~/scanners
wget
    "https://binaries.sonarsource.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-4.2.0.18
unzip sonar-scanner-cli-4.2.0.1873-linux.zip
mv sonar-scanner-4.2.0.1873-linux sonar-scanner
```

Code 32: Setting up the SonarQube client to scan PHP code

A first test run with our whole vulnerability test data can then be started like in listing 33:

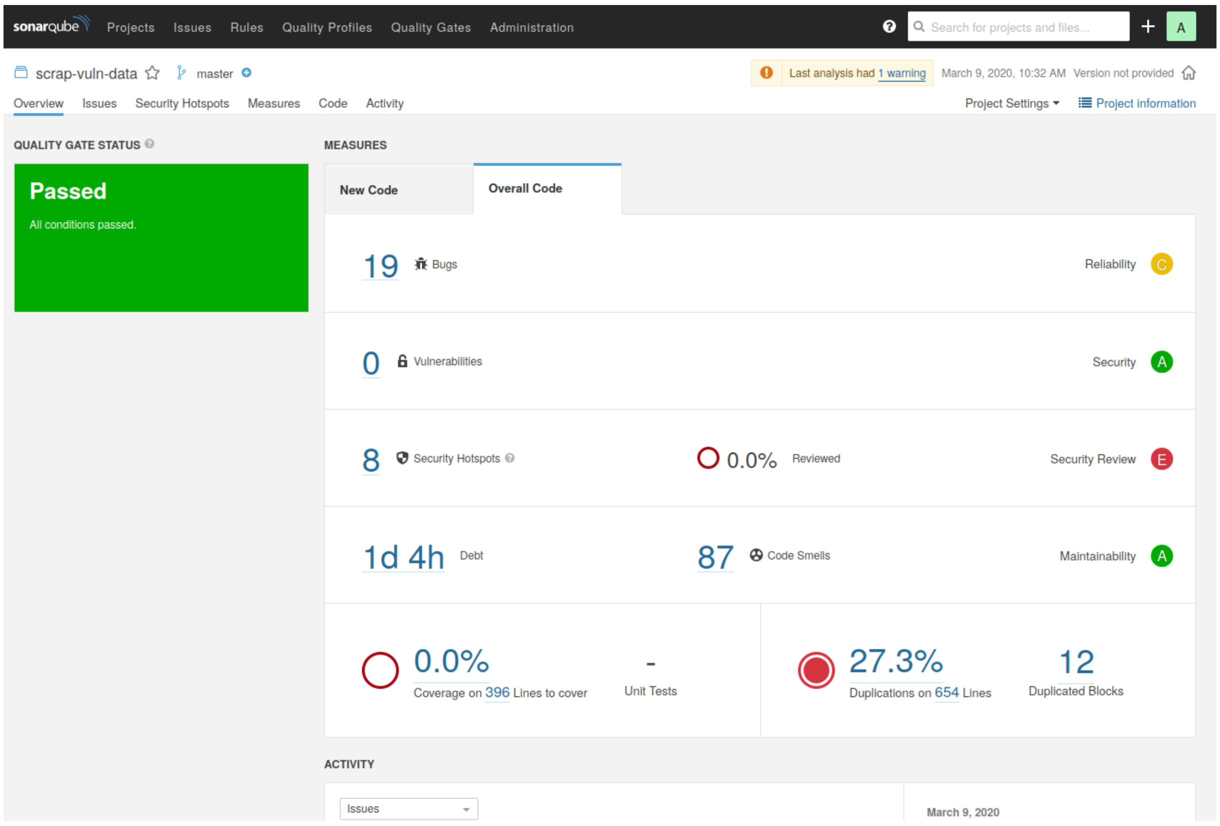
```
scanners/sonar-scanner/bin/sonar-scanner \
  -Dsonar.projectKey=scrap-vuln-data \
  -Dsonar.host.url=http://scrap:9000 \
  -Dsonar.login=839279eea43ef40185cc1f3dacd6d44146b852fb \
  -Dsonar.sources=./scrap-vuln-data
```

Code 33: Scanning the test data set with SonarQube

The project dashboard on the server interface then updates automatically to highlight the results, as seen in screenshot sonarqube02.png, which is also depicted in figure 19.

General assessment: SonarQube is its own mature platform for code analysis, covering many languages and including advanced features. It is advertised not only as a general purpose code quality scanning platform but also as a security focused platform.

This is highlighted especially in the newly introduced *Security Hotspot* reviewing workflow. As screenshot sonarqube03a.png shows, the list of found issues provides filters for many characteristics. Besides the severity of the issues, they are also classified into *code smells*, *bugs*



scrap-vuln-data master
Last analysis had 1 warning March 9, 2020, 10:32 AM Version not provided

QUALITY GATE STATUS
Passed
All conditions passed.

MEASURES

19 Bugs — Reliability C
0 Vulnerabilities — Security A
8 Security Hotspots — 0.0% Reviewed — Security Review E
1d 4h Debt — 87 Code Smells — Maintainability A
0.0% Coverage on 396 Lines to cover — - Unit Tests
27.3% Duplications on 654 Lines — 12 Duplicated Blocks

ACTIVITY
March 9, 2020

Figure 19: SonarQube project dashboard right after a scan of the test data set

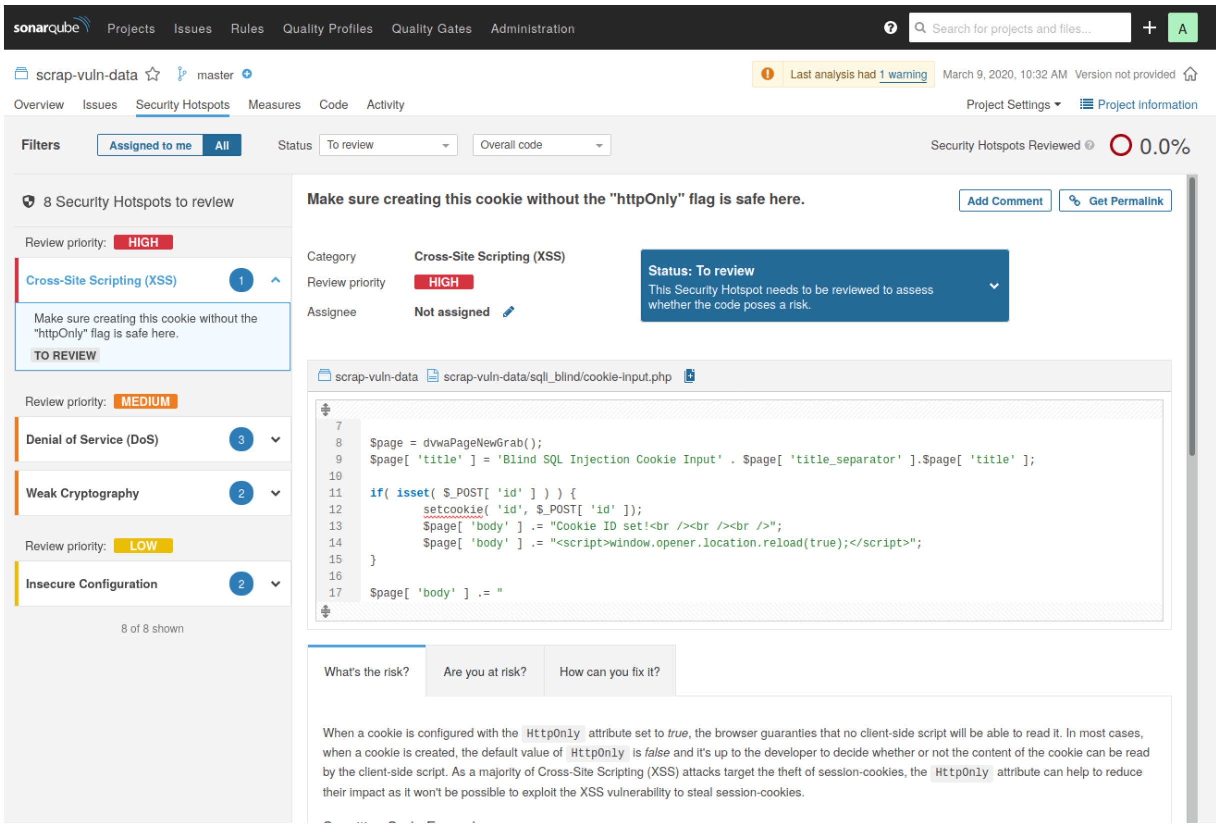
and *vulnerabilities*, and security categories can be provided based on the OWASP Top 10, the SANS Top 25, or available data from the CWE database. Unfortunately no vulnerability was found in the displayed project scan, which was a first evaluative scan of the whole *scrap-vuln-data* directory, as explained in the setup section above.

Nevertheless some *Security Hotspots* could be identified, as screenshot sonarqube03b.png shows (depicted in figure 20). For each hotspot, not only the category and the relevant code sections are highlighted, but also the following three tabs:

1. “What’s the risk?” as shown in sonarqube03c.png
2. “Are you at risk?” as shown in sonarqube03d.png
3. “How can you fix it?” as shown in sonarqube03e.png

This points in the right direction and is in line with research on secure coding education. Especially the UI featuring the relevant code pieces in combination with the tabs explaining the vulnerability and potential mitigations are very promising and should be adopted not only in educational contexts.

Applied to the test data: SonarQube is not tailored towards one-time single file analyses, but rather to incremental analyses of a code base with a thorough project set up. Therefore the evaluation of the test data as-is is rather cumbersome, without initial adaptation.



Make sure creating this cookie without the "httpOnly" flag is safe here.

This Security Hotspot needs to be reviewed to assess whether the code poses a risk.

```

$page = dvwaPageNewGrab();
$page[ 'title' ] = 'Blind SQL Injection Cookie Input' . $page[ 'title_separator' ].$page[ 'title' ];

if( isset( $_POST[ 'id' ] ) ) {
        setcookie( 'id', $_POST[ 'id' ]);
        $page[ 'body' ] .= "Cookie ID set!<br /><br /><br />";
        $page[ 'body' ] .= "<script>window.opener.location.reload(true);</script>";
}

$page[ 'body' ] .= "
```

When a cookie is configured with the `HttpOnly` attribute set to *true*, the browser guarantees that no client-side script will be able to read it. In most cases, when a cookie is created, the default value of `HttpOnly` is *false* and it's up to the developer to decide whether or not the content of the cookie can be read by the client-side script. As a majority of Cross-Site Scripting (XSS) attacks target the theft of session-cookies, the `HttpOnly` attribute can help to reduce their impact as it won't be possible to exploit the XSS vulnerability to steal session-cookies.

Figure 20: The SonarQube dashboard showing Security Hotspots for the scanned project

To make it more feasible for the comparative analysis, a test script has been written in analyse.sh, which can be used for temporary SonarQube project instantiation an scanning of a single files or directories, to retrieve the results and then remove the temporary project. This is also a proof of concept for a workflow as it could be used for SCRAP. The script can also be found in the appendix. With it, the analysis results for the *results.csv* comparison table could then be produced with the commands in listing 34:

```
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/sqli_low.php >
    sqli_low
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/sqli_medium.php >
    sqli_medium
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/sqli_high.php >
    sqli_high
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/xss_r_low.php >
    xss_r_low
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/xss_r_medium.php >
    xss_r_medium
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/xss_r_high.php >
    xss_r_high
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/sqli > sqli
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/sqli_blind >
    sqli_blind
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/xss_r > xss_r
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/xss_s > xss_s
scrap-eval/SonarQube/analyse.sh scrap-temp scrap-vuln-data/xss_d > xss_d
```

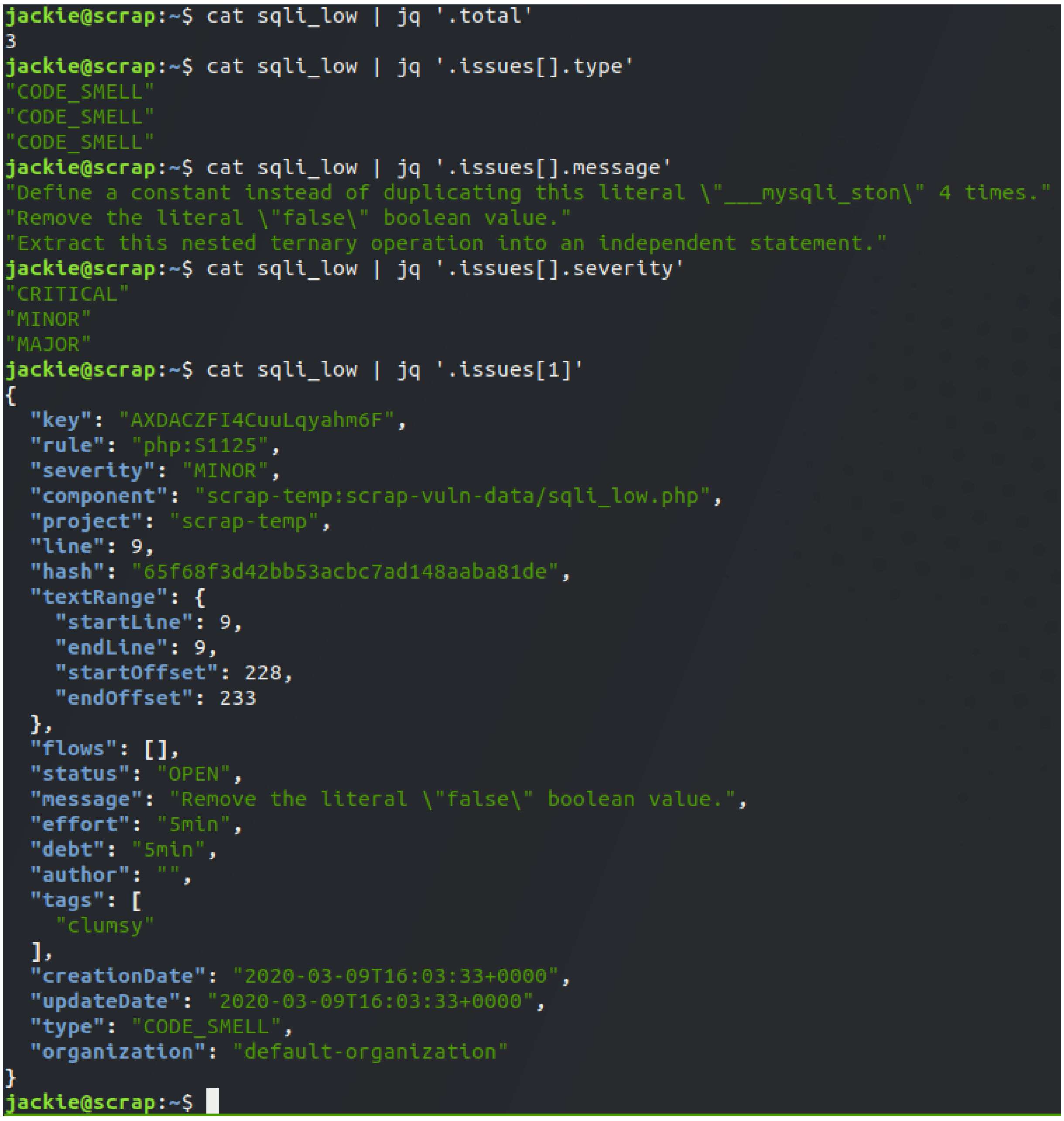
Code 34: Scanning each test data component in a separate SonarQube scan and retrieving the results

Further processing was done with using different `jq` filters to find the number of found issues, the types, messages and full issue descriptions, as the exemplary screenshot sonarqube04.png shows, which is also depicted in figure 21.

The issues found in these results all have been counted as unspecific hits in the results comparison, as the SonarQube Web API does not provide any means to access the newly featured workflow for reviewing *Security Hotspots*.

As the security hotspots where reviewed manually in the web interface for a project scanning the whole *scrap-vuln-data* folder, some vulnerabilites could be found, but none of those that we actually wanted to test for. A common pattern was to find e.g. a XSS vulnerability in the files featuring an SQL injection, a DoS vulnerability in the files featuring the actual XSS, and issues of weak cryptography in a file of the *sqli_blind* set, where the `rand` function was used to sleep a (pseudo)random amount of seconds, as can be seen in the screenshot sonarqube05.png. Therefore, no actual hits have been entered in the comparison table, and only those general findings from the analysis above have been counted.

For the **adaptation in SCRAP**, SonarQube provides a very usable RESTful Web API, wich is documented on the running server under the url http://scrap:9000/web_api (given our standard setup on a host with the name *scrap*).



```
jackie@scrap:~$ cat sqli_low | jq '.total'
3
jackie@scrap:~$ cat sqli_low | jq '.issues[].type'
"CODE_SMELL"
"CODE_SMELL"
"CODE_SMELL"
jackie@scrap:~$ cat sqli_low | jq '.issues[].message'
"Define a constant instead of duplicating this literal \"___mysqli_ston\" 4 times."
"Remove the literal \"false\" boolean value."
"Extract this nested ternary operation into an independent statement."
jackie@scrap:~$ cat sqli_low | jq '.issues[].severity'
"CRITICAL"
"MINOR"
"MAJOR"
jackie@scrap:~$ cat sqli_low | jq '.issues[1]'
{
  "key": "AXDACZFI4CuuLqyahm6F",
  "rule": "php:S1125",
  "severity": "MINOR",
  "component": "scrap-temp:scrap-vuln-data/sqli_low.php",
  "project": "scrap-temp",
  "line": 9,
  "hash": "65f68f3d42bb53acbc7ad148aaba81de",
  "textRange": {
    "startLine": 9,
    "endLine": 9,
    "startOffset": 228,
    "endOffset": 233
  },
  "flows": [],
  "status": "OPEN",
  "message": "Remove the literal \"false\" boolean value.",
  "effort": "5min",
  "debt": "5min",
  "author": "",
  "tags": [
    "clumsy"
  ],
  "creationDate": "2020-03-09T16:03:33+0000",
  "updateDate": "2020-03-09T16:03:33+0000",
  "type": "CODE_SMELL",
  "organization": "default-organization"
}
jackie@scrap:~$
```

Figure 21: Filtering the SonarQube scan results with jq

Getting issues for a component (project, directory, file, ...) is therefore quite easy, given a user token was created (alternatively, user credentials can be used with HTTP Basic authentication, but the use of a revokable token should be the preferred method, especially if this is documented publicly, like in the case of this thesis). A corresponding `curl` call combined with a `jq` filter is displayed in listing 35:

```
curl -s -u bb3fbae1a08695d69d596debef18b12d54936cb6: \
scrap:9000/api/issues/search?componentKeys=scrap-vuln-data:scrap-vuln-data/sqli_low.php
    | jq .
```

Code 35: Using the SonarQube web API to retrieve scan results

While very promising, the new workflow to review *Security Hotspots* was only introduced in the current version 8.2, which was announced on February 26th, 2020. The Web API did not provide any means to access the security hotspots. The next version to be released (currently 8.3.) can be tested under https://next.sonarqube.com, and a look at its Web API shows that there are already several endpoints that are not yet available in version 8.2. Nevertheless also in 8.3 there is not hint towards reviewing the security hotspots. Probably this feature will be implemented in a future version yet to come. For the current evaluation for SCRAP, this means that SonarQube is not as usable as some of the other tools tested, despite its otherwise intriguing web interface and the Web API. But it has to be noted, that the features implemented in the web interface point in the right direction and fulfils many of the criteria that have been proposed by the research on secure coding education, as reviewd in chapter 2.

There is also documentation available for writing own coding rules, which would be necessary to improve the vulnerability detection. But as the SonarQube project seems to progress steadily, the most efficient thing would be to rather do a re-evaluation of its coming major releases, than "duct taping" the current release to SCRAP.

Résumé

To compare all those 7 tools, every tool was tested on every of the single vulnerability files in the root of the test data folder, as well as on the 5 folders containing all code for the mentioned vulnerability types. This makes for 11 separate tests for each tool.

For every test all explicit findings of a vulnerability were counted as a *direct hit*. If one of the vulnerabilities was found but not labelled as vulnerability / security issue, it was counted as *vulnerabilities hit as unspecific*. All other findings were counted as *unspecific hits*.

The results of all those tests were compiled into the results.csv. table. To generate some visualisation of the findings, the Python script makeplots.py (available also in the appendix) was created, which facilitates pandas and matplotlib to filter the data and generate the following 5 plots:

1. scanner_comparison_overall.png

2. scanner_comparison_sqli_blind.png

3. scanner_comparison_sqli.png
4. scanner_comparison_xss_r.png
5. scanner_comparison_xss_s.png

To come to the final résumé, lets have a look at the overall comparison as depicted in figure 22.

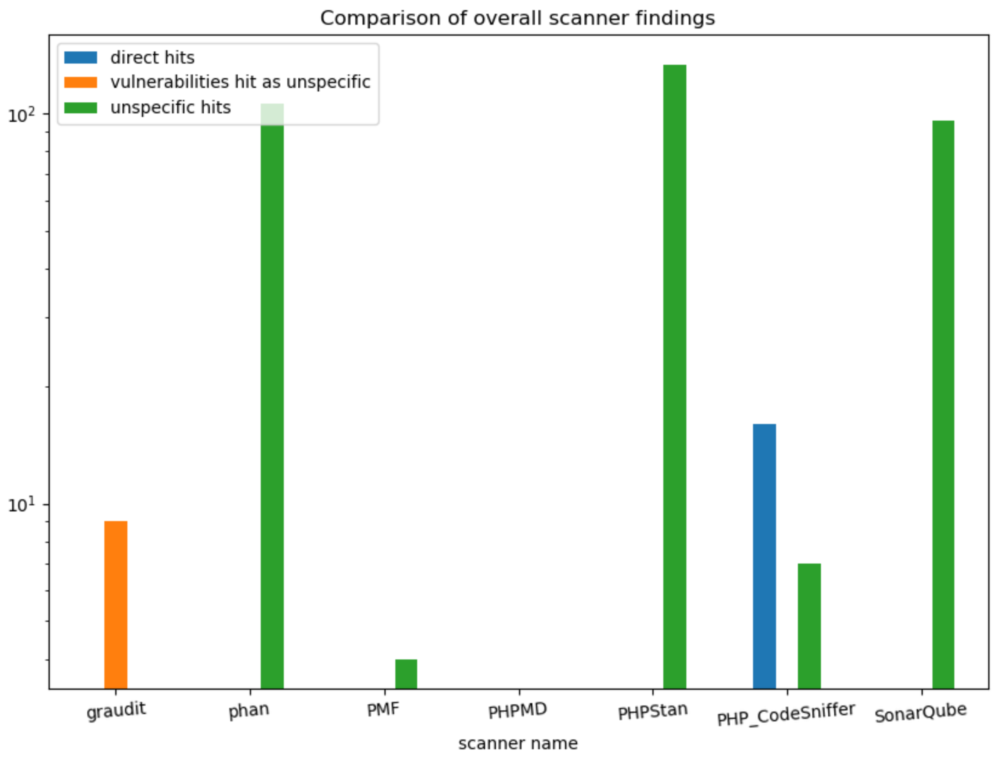


Figure 22: Overall comparison of scanner tools

As we can see, only *PHP_CodeSniffer* found specific vulnerabilities. Here we have to note, that this is only because we use it with the *phpcs-security-audit v2* rule set instead of its original own rule set. And while *phan*, *PHPStan* and *SonarQube* found significantly more general issues, most of them are due to linting issues and the fact that the test data is a subset of the whole *DVWA* application, which leads to many uninitialised references.

While SonarQube is promoted also with a security focus, at least for the PHP test data set provided here, the results are not promising. Nevertheless its framework and web user interface seem promising and it is built to be integrated into other workflows. Yet, for an adaptation in context of the (time-limited) SCRAP project, the resources needed for adoption outweigh the curiosity and applicability. As stated in the closer analysis of *SonarQube* a future release might prove more promising in terms of code security, as the new workflow to review *Security Hotspots* was only introduced in its most recent release.

Also notable is that *graudit* finds the SQL injection vulnerabilites, but does not provide any information on what it actually found. After all, it is just a simple regular-expression based grepper tool. On the other side, it is quite easy to adapt such a tool, by providing additional regex patterns. In the case of *PHP Malware Finder* this lead to the realisation that in our case we could just facilitate *yara* as a tool that is already made for finding vulnerabilities based on signatures defined in YARA rules.

For those reasons the SCRAP prototype includes two exemplary adaptations of the F/LOSS analysis tools found in this evaluation:

- *PHP_CodeSniffer* with the *phpcs-security-audit v2*
- *yara* with the *PHP Malware Finder* rule set, extended with own rules

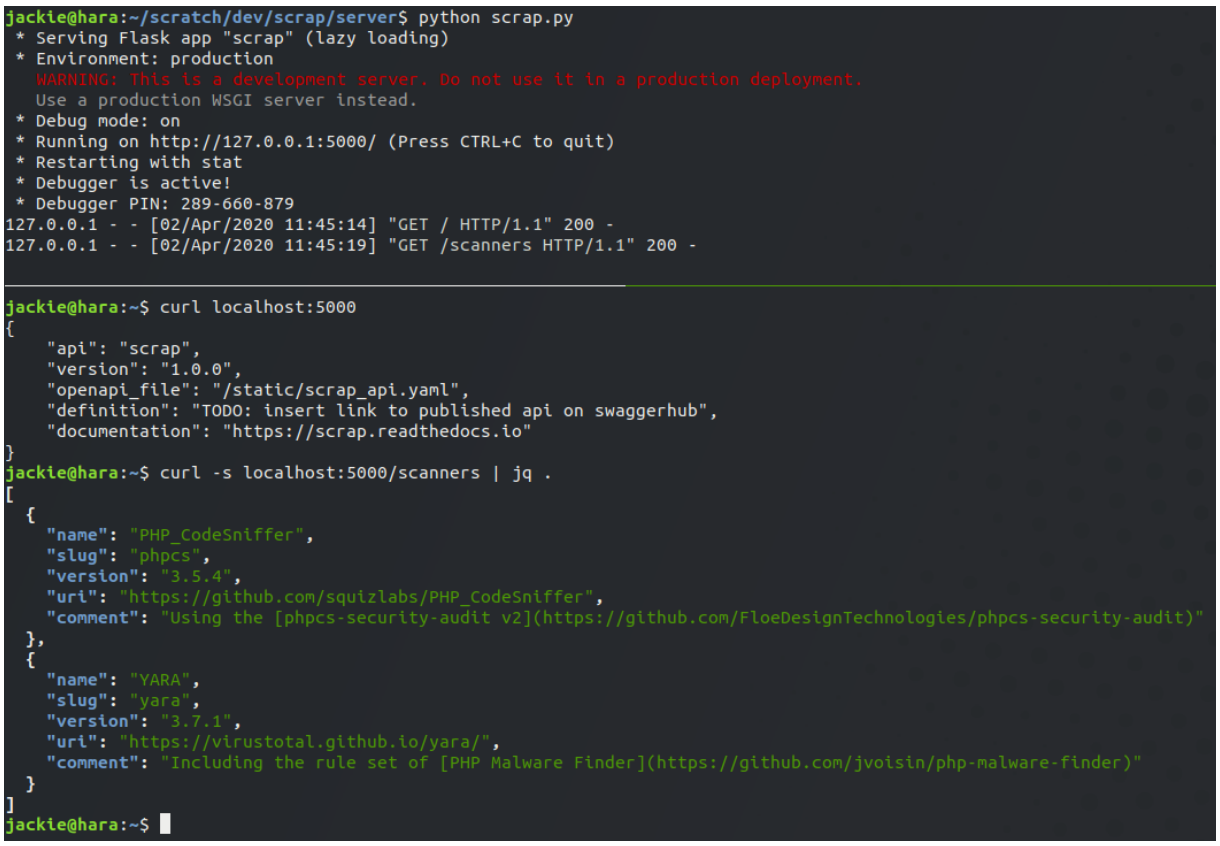
5.3.3 SCRAP

The final SCRAP prototype consists of the web service, also called the SCRAP API server, and the web client, or SCRAP UI. For both the source code is available on GitLab in the scrap-api-server and scrap-client repositories, as well as in their copies on the accompanying data disc. Additionally I plan to operate a demo server for the web service and the client, which will be available under the project site https://scrap.tantemalkah.at.

Before evaluating the prototype system in its whole, let us take a look at at how the two components work and look like. Figure 23 shows two instances of `bash` (tiled in a *tmux* window). In the upper pane we see how the server gets started (in the development environment) and how the first two requests come in. In the lower pane we see those two requests initiated by two simple `curl` calls, one time to the root endpoint of the API and then to the */scanners* endpoint. The latter is filtered through `jq` for a more readable and marked up representation.

Figure 24 shows how the list of available explanations are retrieved, which are very few and mostly placeholder explanations in this prototype setting. Figure 25 then shows one specific explanation. This will be revisited later when we come to the web UI, which provides a meaningful visualisation of issues and explanations. Figure 26 shows what happens when we access a non-existing explanations. As expected we receive a 404 HTTP response from the web service, but additionally the response body contains a meaningful JSON representation of what this error is about. This is defined in the API specification and works similar when we try to access non-existing scans, issues or files.

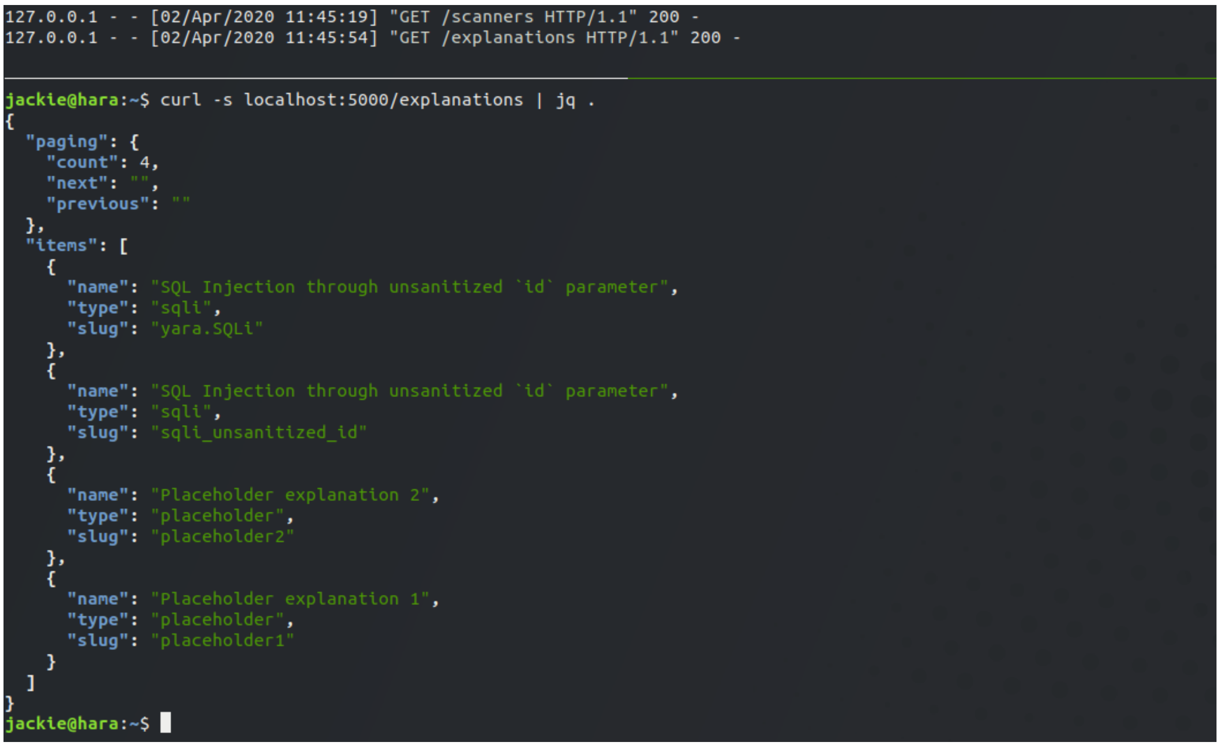
In a similar way we also get an 401 (Not Authorized) error response when we try to access the */scans* endpoint without a valid API user and API key, as shown in 27. Figure 28 then shows a successful response when we use the public API user and API key, as they are configured in the test environment. Whether there is a public account and how it is called is configured in the *config.py* file, and a corresponding entry in the *apikeys* table in the database has to exist. What we also see in those two screenshots is, that the server, when in debug mode, logs the user names for all authenticated and unauthenticated requests to all endpoints that are covered by



```
jackie@hara:~/scratch/dev/scrap/server$ python scrap.py
 * Serving Flask app "scrap" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 289-660-879
127.0.0.1 - - [02/Apr/2020 11:45:14] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2020 11:45:19] "GET /scanners HTTP/1.1" 200 -
```

```
jackie@hara:~$ curl localhost:5000
{
    "api": "scrap",
    "version": "1.0.0",
    "openapi_file": "/static/scrap_api.yaml",
    "definition": "TODO: insert link to published api on swaggerhub",
    "documentation": "https://scrap.readthedocs.io"
}
jackie@hara:~$ curl -s localhost:5000/scanners | jq .
[
  {
    "name": "PHP_CodeSniffer",
    "slug": "phpcs",
    "version": "3.5.4",
    "uri": "https://github.com/squizlabs/PHP_CodeSniffer",
    "comment": "Using the [phpcs-security-audit v2](https://github.com/FloeDesignTechnologies/phpcs-security-audit)"
  },
  {
    "name": "YARA",
    "slug": "yara",
    "version": "3.7.1",
    "uri": "https://virustotal.github.io/yara/",
    "comment": "Including the rule set of [PHP Malware Finder](https://github.com/jvoisin/php-malware-finder)"
  }
]
jackie@hara:~$
```

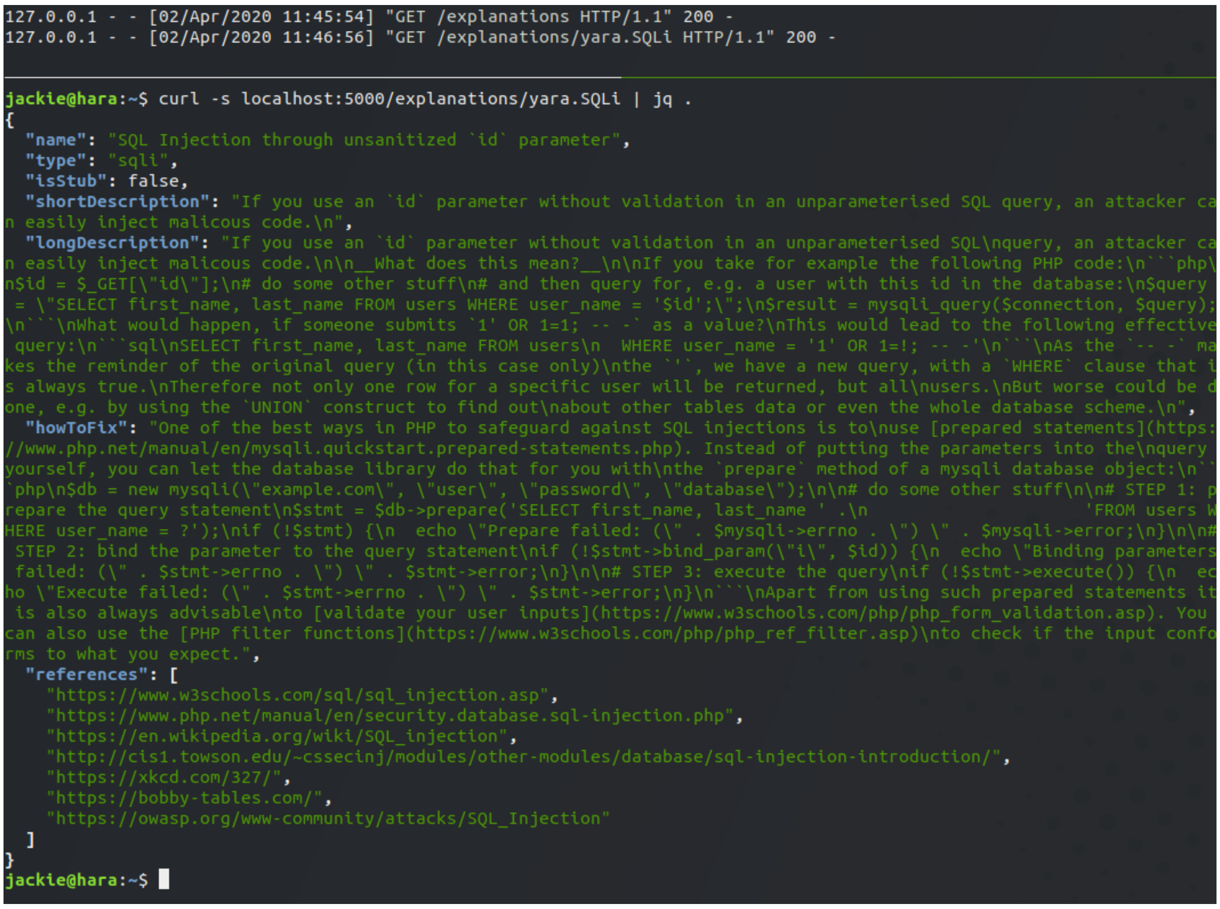
Figure 23: Starting the SCRAP API server and accessing its root and /scanners endpoints



```
127.0.0.1 - - [02/Apr/2020 11:45:19] "GET /scanners HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2020 11:45:54] "GET /explanations HTTP/1.1" 200 -
```

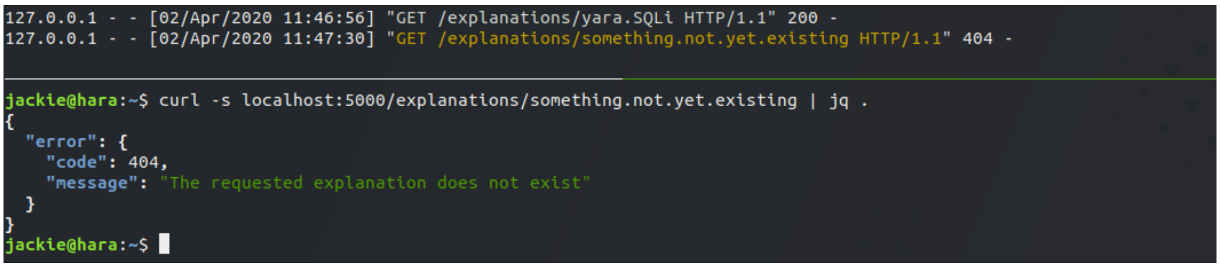
```
jackie@hara:~$ curl -s localhost:5000/explanations | jq .
{
  "paging": {
    "count": 4,
    "next": "",
    "previous": ""
  },
  "items": [
    {
      "name": "SQL Injection through unsanitized `id` parameter",
      "type": "sqli",
      "slug": "yara.SQLi"
    },
    {
      "name": "SQL Injection through unsanitized `id` parameter",
      "type": "sqli",
      "slug": "sqli_unsanitized_id"
    },
    {
      "name": "Placeholder explanation 2",
      "type": "placeholder",
      "slug": "placeholder2"
    },
    {
      "name": "Placeholder explanation 1",
      "type": "placeholder",
      "slug": "placeholder1"
    }
  ]
}
jackie@hara:~$
```

Figure 24: Accessing the /explanations endpoint with curl



```
127.0.0.1 - - [02/Apr/2020 11:45:54] "GET /explanations HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2020 11:46:56] "GET /explanations/yara.SQLi HTTP/1.1" 200 -

jackie@hara:~$ curl -s localhost:5000/explanations/yara.SQLi | jq .
{
  "name": "SQL Injection through unsanitized `id` parameter",
  "type": "sqli",
  "isStub": false,
  "shortDescription": "If you use an `id` parameter without validation in an unparameterised SQL query, an attacker ca
n easily inject malicous code.\n",
  "longDescription": "If you use an `id` parameter without validation in an unparameterised SQL\nquery, an attacker ca
n easily inject malicous code.\n\n__What does this mean?__\n\nIf you take for example the following PHP code:\n```php\
n$id = $_GET[\"id\"];\n# do some other stuff\n# and then query for, e.g. a user with this id in the database:\n$query
 = \"SELECT first_name, last_name FROM users WHERE user_name = '$id';\";\n$result = mysqli_query($connection, $query);
\n```\nWhat would happen, if someone submits `1' OR 1=1; -- -` as a value?\nThis would lead to the following effective
 query:\n```sql\nSELECT first_name, last_name FROM users\n  WHERE user_name = '1' OR 1=!; -- -'\n```\nAs the `-- -` ma
kes the reminder of the original query (in this case only)\nthe `'`, we have a new query, with a `WHERE` clause that i
s always true.\nTherefore not only one row for a specific user will be returned, but all\nusers.\nBut worse could be d
one, e.g. by using the `UNION` construct to find out\nabout other tables data or even the whole database scheme.\n",
  "howToFix": "One of the best ways in PHP to safeguard against SQL injections is to\nuse [prepared statements](https:
//www.php.net/manual/en/mysqli.quickstart.prepared-statements.php). Instead of putting the parameters into the\nquery
yourself, you can let the database library do that for you with\nthe `prepare` method of a mysqli database object:\n``
`php\n$db = new mysqli(\"example.com\", \"user\", \"password\", \"database\");\n\n# do some other stuff\n\n# STEP 1: p
repare the query statement\n$stmt = $db->prepare('SELECT first_name, last_name ' .\n                        'FROM users W
HERE user_name = ?');\nif (!$stmt) {\n  echo \"Prepare failed: (\" . $mysqli->errno . \") \" . $mysqli->error;\n}\n\n#
 STEP 2: bind the parameter to the query statement\nif (!$stmt->bind_param(\"i\", $id)) {\n  echo \"Binding parameters
 failed: (\" . $stmt->errno . \") \" . $stmt->error;\n}\n\n# STEP 3: execute the query\nif (!$stmt->execute()) {\n  ec
ho \"Execute failed: (\" . $stmt->errno . \") \" . $stmt->error;\n}\n```\nApart from using such prepared statements it
 is also always advisable\nto [validate your user inputs](https://www.w3schools.com/php/php_form_validation.asp). You
can also use the [PHP filter functions](https://www.w3schools.com/php/php_ref_filter.asp)\nto check if the input confo
rms to what you expect.",
  "references": [
    "https://www.w3schools.com/sql/sql_injection.asp",
    "https://www.php.net/manual/en/security.database.sql-injection.php",
    "https://en.wikipedia.org/wiki/SQL_injection",
    "http://cis1.towson.edu/~cssecinj/modules/other-modules/database/sql-injection-introduction/",
    "https://xkcd.com/327/",
    "https://bobby-tables.com/",
    "https://owasp.org/www-community/attacks/SQL_Injection"
  ]
}
jackie@hara:~$ 
```

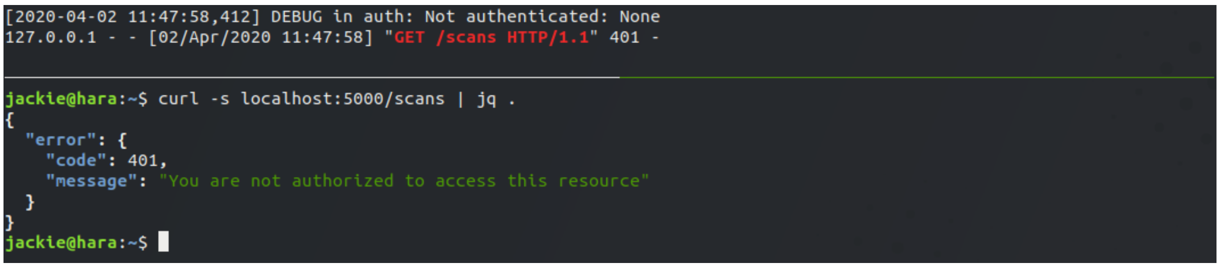
Figure 25: Accessing a single explanation with curl



```
127.0.0.1 - - [02/Apr/2020 11:46:56] "GET /explanations/yara.SQLi HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2020 11:47:30] "GET /explanations/something.not.yet.existing HTTP/1.1" 404 -

jackie@hara:~$ curl -s localhost:5000/explanations/something.not.yet.existing | jq .
{
  "error": {
    "code": 404,
    "message": "The requested explanation does not exist"
  }
}
jackie@hara:~$ 
```

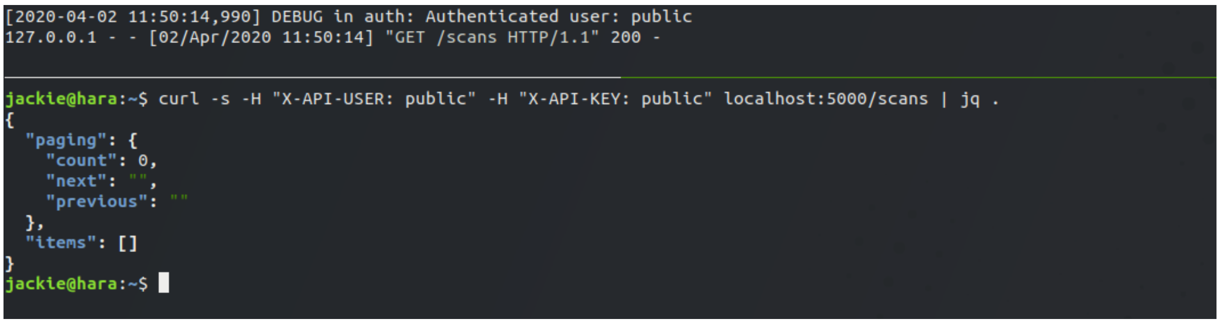
Figure 26: Accessing a non-existing explanation with curl

the *auth* tag in the API specification. While the public user has no scans available yet, figure 29 shows a request from a test user who already uploaded several files to be scanned. Figure 30 shows how this user can retrieve information for a specific scan.



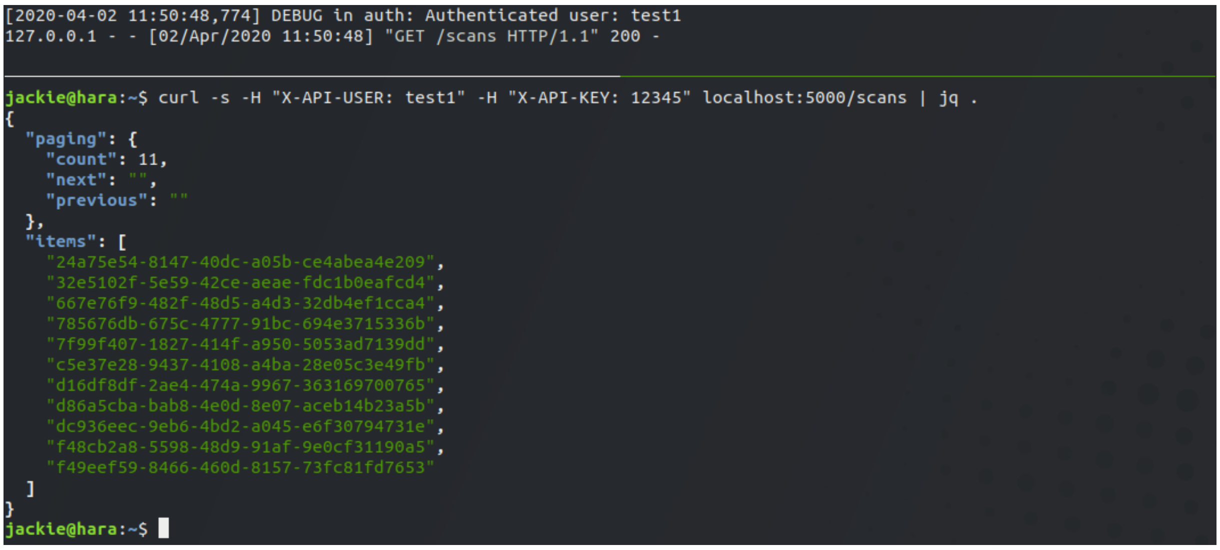
```
[2020-04-02 11:47:58,412] DEBUG in auth: Not authenticated: None
127.0.0.1 - - [02/Apr/2020 11:47:58] "GET /scans HTTP/1.1" 401 -

jackie@hara:~$ curl -s localhost:5000/scans | jq .
{
  "error": {
    "code": 401,
    "message": "You are not authorized to access this resource"
  }
}
jackie@hara:~$
```

Figure 27: Accessing the /scans endpoint without an API key with curl



```
[2020-04-02 11:50:14,990] DEBUG in auth: Authenticated user: public
127.0.0.1 - - [02/Apr/2020 11:50:14] "GET /scans HTTP/1.1" 200 -

jackie@hara:~$ curl -s -H "X-API-USER: public" -H "X-API-KEY: public" localhost:5000/scans | jq .
{
  "paging": {
    "count": 0,
    "next": "",
    "previous": ""
  },
  "items": []
}
jackie@hara:~$
```

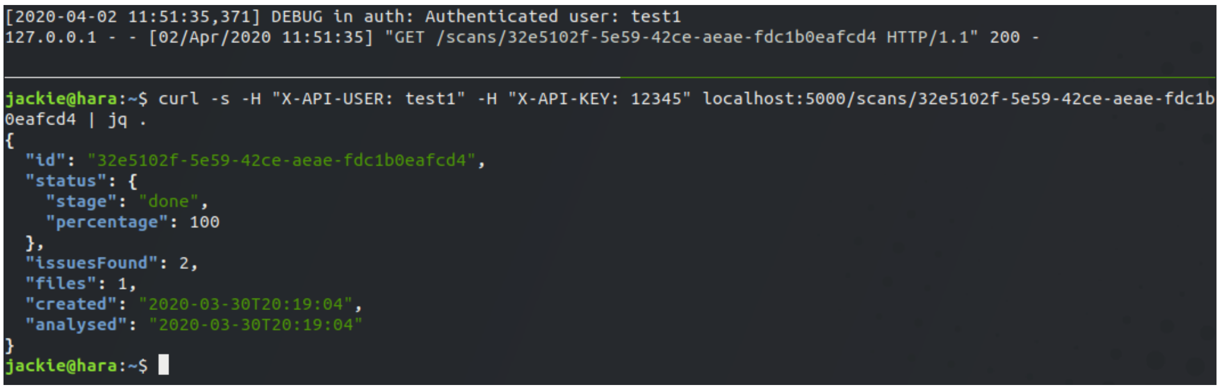
Figure 28: Accessing the /scans endpoint with the public user and API key with curl



```
[2020-04-02 11:50:48,774] DEBUG in auth: Authenticated user: test1
127.0.0.1 - - [02/Apr/2020 11:50:48] "GET /scans HTTP/1.1" 200 -

jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans | jq .
{
  "paging": {
    "count": 11,
    "next": "",
    "previous": ""
  },
  "items": [
    "24a75e54-8147-40dc-a05b-ce4abea4e209",
    "32e5102f-5e59-42ce-aeae-fdc1b0eafcd4",
    "667e76f9-482f-48d5-a4d3-32db4ef1cca4",
    "785676db-675c-4777-91bc-694e3715336b",
    "7f99f407-1827-414f-a950-5053ad7139dd",
    "c5e37e28-9437-4108-a4ba-28e05c3e49fb",
    "d16df8df-2ae4-474a-9967-363169700765",
    "d86a5cba-bab8-4e0d-8e07-aceb14b23a5b",
    "dc936eec-9eb6-4bd2-a045-e6f30794731e",
    "f48cb2a8-5598-48d9-91af-9e0cf31190a5",
    "f49eef59-8466-460d-8157-73fc81fd7653"
  ]
}
jackie@hara:~$
```

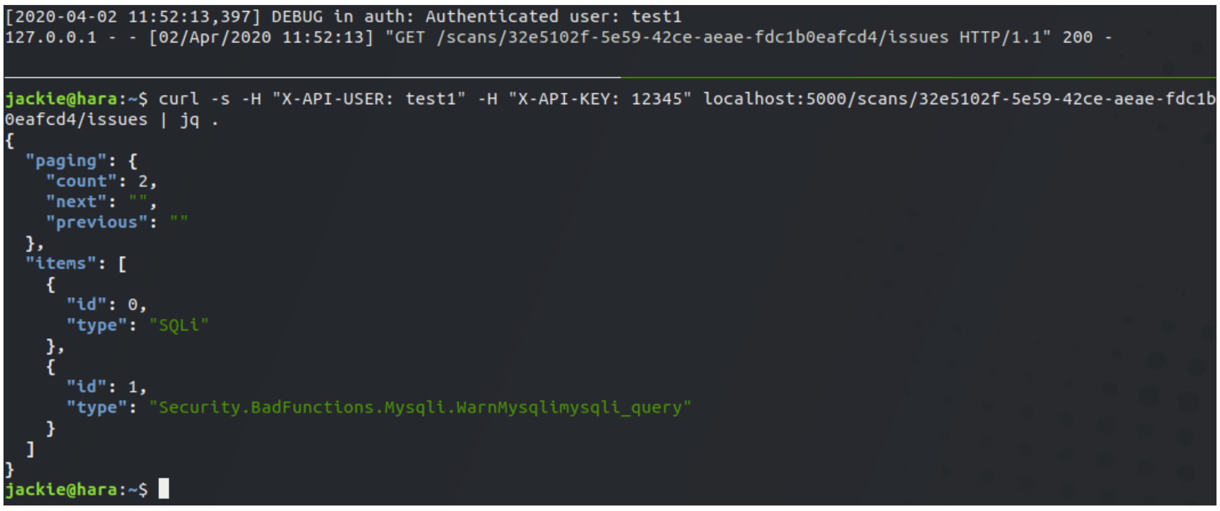
Figure 29: Accessing the scan listing of a non-public test user with curl

In the response showing a specific scan result, we see that the scan is already completed and that scanned project consists of 1 file. In this 1 file the scanners found 2 issues overall. Figure 31 shows how the list of those issues is retrieved and in figure 32 we see the contents of the first of those two issues as a response to the specific issue request.



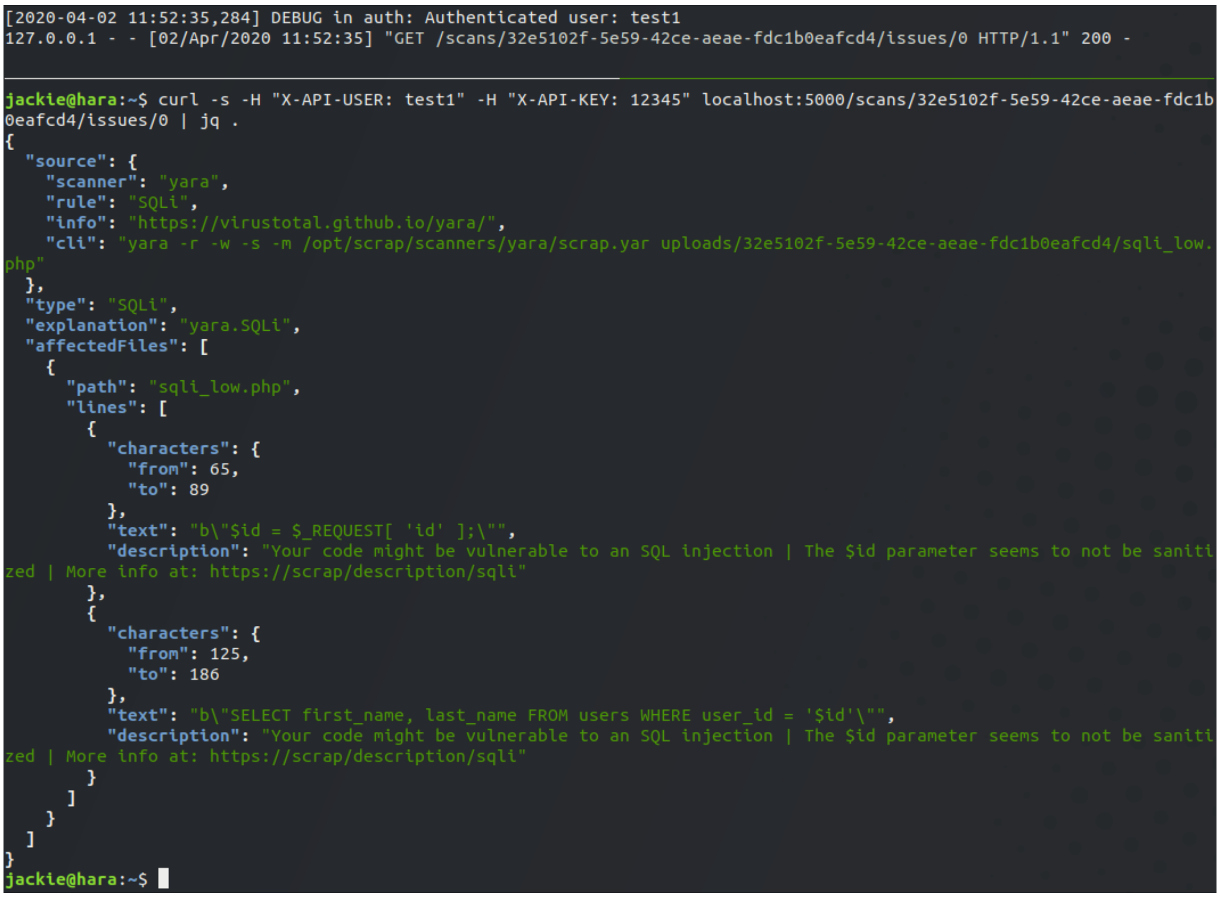
```
[2020-04-02 11:51:35,371] DEBUG in auth: Authenticated user: test1
127.0.0.1 - - [02/Apr/2020 11:51:35] "GET /scans/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4 HTTP/1.1" 200 -

jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans/32e5102f-5e59-42ce-aeae-fdc1b
0eafcd4 | jq .
{
  "id": "32e5102f-5e59-42ce-aeae-fdc1b0eafcd4",
  "status": {
    "stage": "done",
    "percentage": 100
  },
  "issuesFound": 2,
  "files": 1,
  "created": "2020-03-30T20:19:04",
  "analysed": "2020-03-30T20:19:04"
}
jackie@hara:~$
```

Figure 30: Accessing a specific scan with curl



```
[2020-04-02 11:52:13,397] DEBUG in auth: Authenticated user: test1
127.0.0.1 - - [02/Apr/2020 11:52:13] "GET /scans/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4/issues HTTP/1.1" 200 -

jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans/32e5102f-5e59-42ce-aeae-fdc1b
0eafcd4/issues | jq .
{
  "paging": {
    "count": 2,
    "next": "",
    "previous": ""
  },
  "items": [
    {
      "id": 0,
      "type": "SQLi"
    },
    {
      "id": 1,
      "type": "Security.BadFunctions.Mysqli.WarnMysqlimysqli_query"
    }
  ]
}
jackie@hara:~$
```

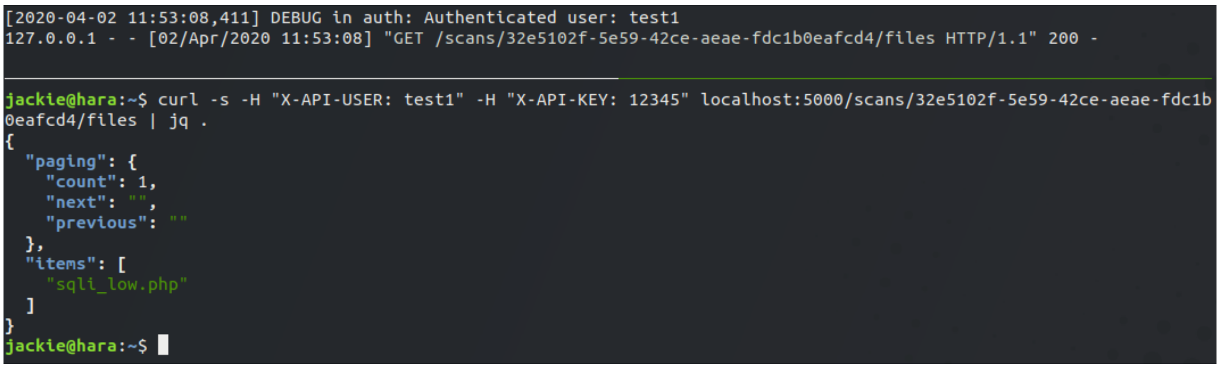
Figure 31: Accessing the issue listing for a specific scan



```
[2020-04-02 11:52:35,284] DEBUG in auth: Authenticated user: test1
127.0.0.1 - - [02/Apr/2020 11:52:35] "GET /scans/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4/issues/0 HTTP/1.1" 200 -

jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans/32e5102f-5e59-42ce-aeae-fdc1b
0eafcd4/issues/0 | jq .
{
  "source": {
    "scanner": "yara",
    "rule": "SQLi",
    "info": "https://virustotal.github.io/yara/",
    "cli": "yara -r -w -s -m /opt/scrap/scanners/yara/scrap.yar uploads/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4/sqli_low.
php"
  },
  "type": "SQLi",
  "explanation": "yara.SQLi",
  "affectedFiles": [
    {
      "path": "sqli_low.php",
      "lines": [
        {
          "characters": {
            "from": 65,
            "to": 89
          },
          "text": "b\"$id = $_REQUEST[ 'id' ];\"",
          "description": "Your code might be vulnerable to an SQL injection | The $id parameter seems to not be saniti
zed | More info at: https://scrap/description/sqli"
        },
        {
          "characters": {
            "from": 125,
            "to": 186
          },
          "text": "b\"SELECT first_name, last_name FROM users WHERE user_id = '$id'\"",
          "description": "Your code might be vulnerable to an SQL injection | The $id parameter seems to not be saniti
zed | More info at: https://scrap/description/sqli"
        }
      ]
    }
  ]
}
jackie@hara:~$ 
```

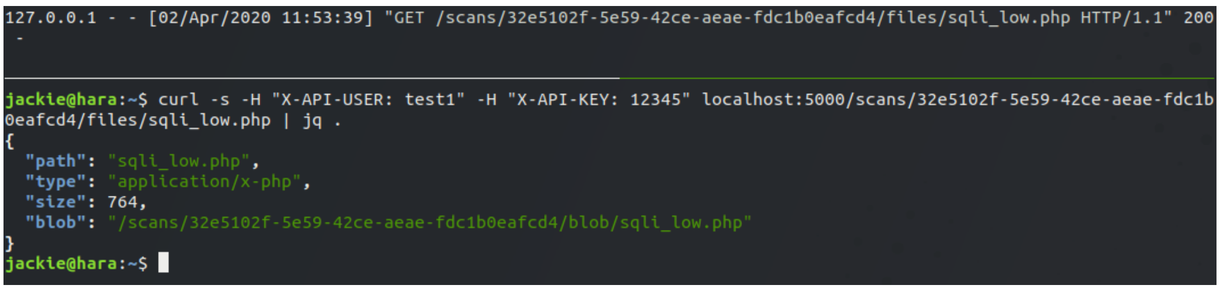
Figure 32: Accessing one specific issue of a scan

We have already seen above how an explanation can be retrieved, which might be linked to in a specific issue. If we also want to retrieve the file's content we can use the *files* and *blob* sub endpoints of a specific scan, as figures 33, 34 and 35 show. This can be used by an UI to represent the issue with rich markup of the source code and to point exactly to the positions where the vulnerability hits the application.



```
[2020-04-02 11:53:08,411] DEBUG in auth: Authenticated user: test1
127.0.0.1 - - [02/Apr/2020 11:53:08] "GET /scans/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4/files HTTP/1.1" 200 -

jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans/32e5102f-5e59-42ce-aeae-fdc1b
0eafcd4/files | jq .
{
  "paging": {
    "count": 1,
    "next": "",
    "previous": ""
  },
  "items": [
    "sqli_low.php"
  ]
}
jackie@hara:~$
```

Figure 33: Accessing the list of files for a specific scan



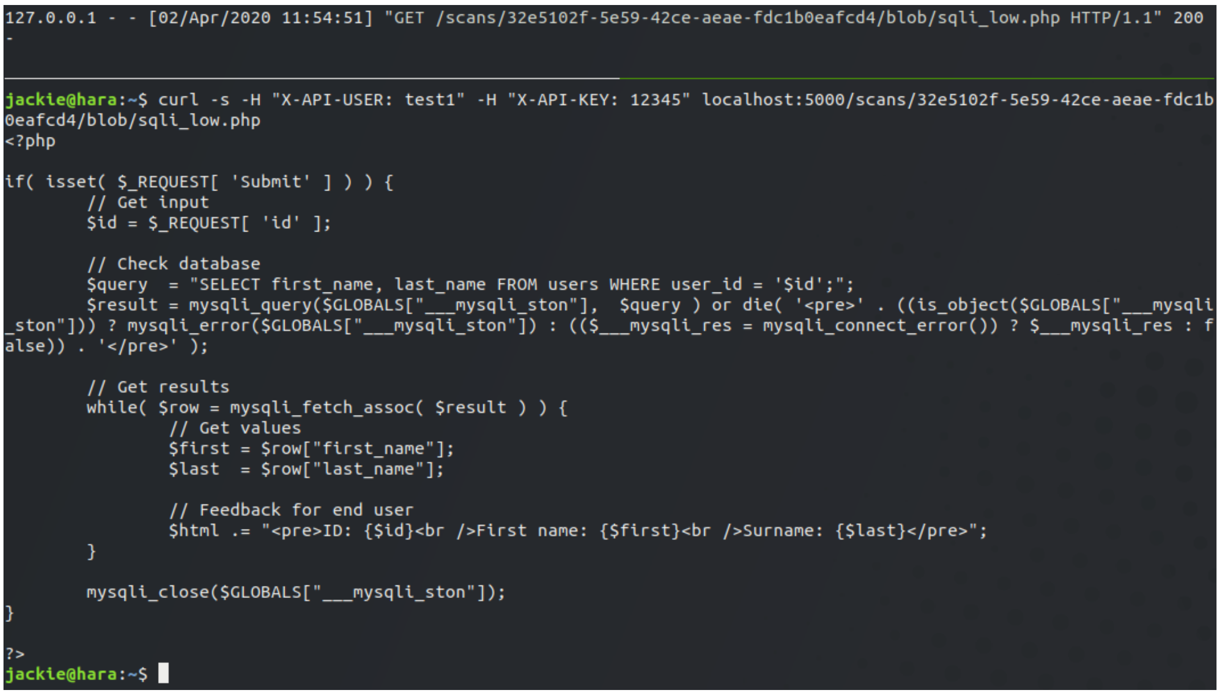
```
127.0.0.1 - - [02/Apr/2020 11:53:39] "GET /scans/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4/files/sqli_low.php HTTP/1.1" 200
 -

jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans/32e5102f-5e59-42ce-aeae-fdc1b
0eafcd4/files/sqli_low.php | jq .
{
  "path": "sqli_low.php",
  "type": "application/x-php",
  "size": 764,
  "blob": "/scans/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4/blob/sqli_low.php"
}
jackie@hara:~$
```

Figure 34: Accessing the meta information for a specific file in a scan

What is finally missing here is how to upload files and initiate an new scan. This is shown in figures 36 and 37, which facilitates the Postman API client, which was used throughout the development of the prototypes much more than the CLI command `curl`, as working with *Postman* is less tedious and more productive. In both of those screenshots we see a POST requests to the */scans* endpoint, both times with the API user *test1*. The user and API key headers are set in the *Headers* tab that is not shown in the screenshot. What we see is the *Body* tab, where the request body is set up. In the first screenshot we send the *sqli_low.php* file and also set the *scanners* parameter to `yara` and the *withIssues* parameter to `true`. This results in a scan, that only facilitates the *YARAScanner* and appends the found issues right in the response of the scan object. The same could also be accomplished with `curl` from the command line, with the following command in listing 36, given one is inside the folder where the *sqli_low.php* file resides.

```
curl -H "X-API-USER:␣test1" -H "X-API-KEY:␣12345" -F scanner=yara \
  -F withIssues=true -F file=@sqli_low.php https://127.0.0.1:5000/scans
```

Code 36: Uploading a file to scan with curl



```
127.0.0.1 - - [02/Apr/2020 11:54:51] "GET /scans/32e5102f-5e59-42ce-aeae-fdc1b0eafcd4/blob/sqli_low.php HTTP/1.1" 200
-

jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans/32e5102f-5e59-42ce-aeae-fdc1b
0eafcd4/blob/sqli_low.php
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
        // Get input
        $id = $_REQUEST[ 'id' ];

        // Check database
        $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die( '<pre>' . ((is_object($GLOBALS["___mysqli
_ston"])) ? mysqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res = mysqli_connect_error()) ? $___mysqli_res : f
alse)) . '</pre>' );

        // Get results
        while( $row = mysqli_fetch_assoc( $result ) ) {
                // Get values
                $first = $row["first_name"];
                $last  = $row["last_name"];

                // Feedback for end user
                $html .= "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
        }

        mysqli_close($GLOBALS["___mysqli_ston"]);
}

?>
jackie@hara:~$ 
```

Figure 35: Retrieving a specific file from a scan

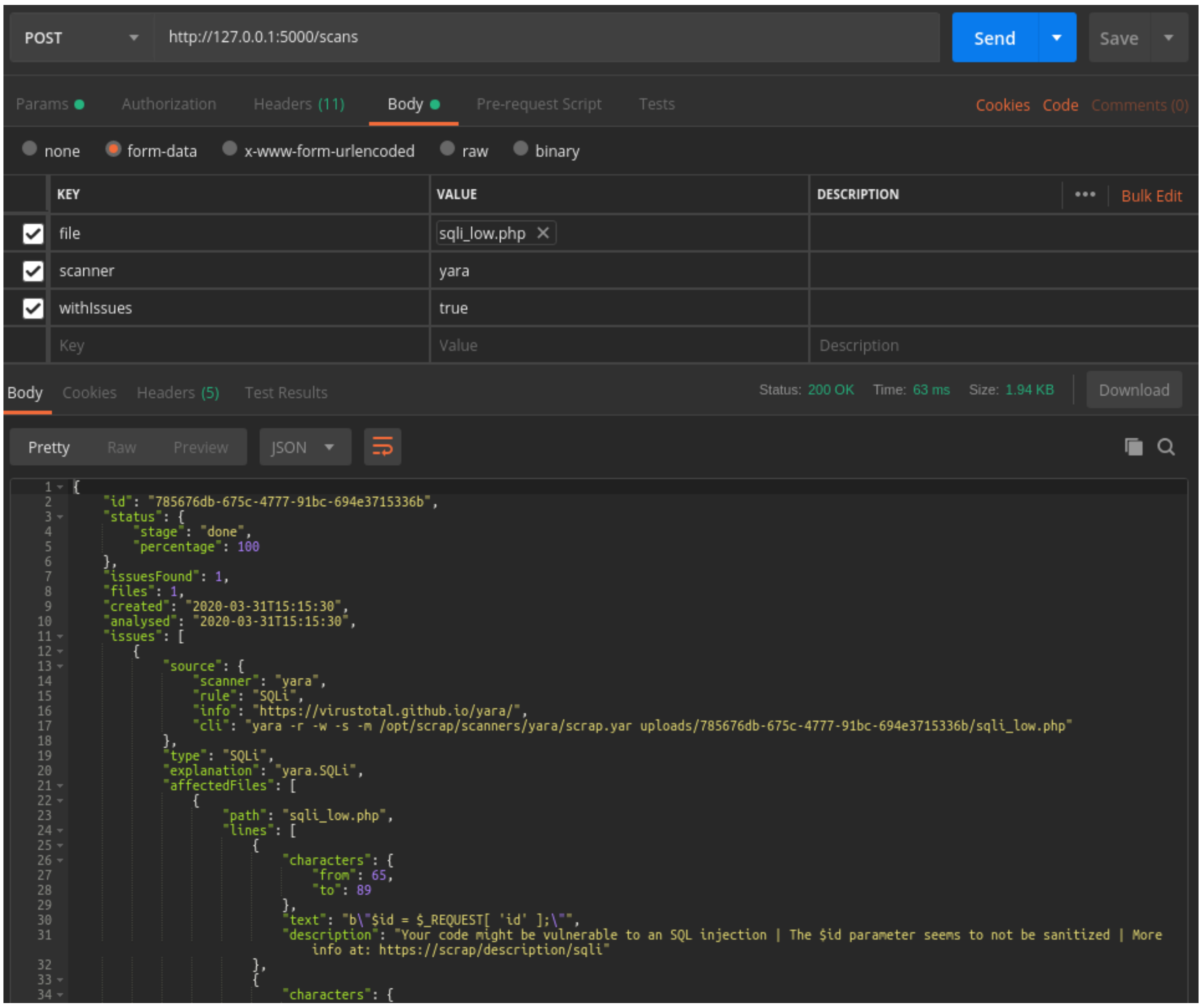
In the second screenshot only the *sqli_low.php* file is sent, without the additional parameters. This results in a scan facilitating all available scanners but not returning the results immediately.

What we have seen so far covers the API and the server component. When it comes to the web UI, the most interesting part is how the issues and explanations are visualized. When we choose a scan from our scan listing, as displayed in figure 38, we get to the *Scan* view of the client, shown in figure 39. This lists the scans meta information and the number of files scanned and issues found, just as we saw it above in the output retrieved with `curl`. Then a list of issues, if any where found, is provided, where the user can click on the single issues. If no issues have been found the user gets a nice green check mark instead of the red warning sign, and accordingly there is no list of issues to choose from.

When an issue is selected, as in this example the *SQLi* issue found by *yara*, the corresponding parts in the file are displayed with three tabs for the explanation below it. In the prototype version the text field containing the file contents only shows the matched piece of code with its offset in the file and an added comment on what the problem is. In a fully developed version here the whole file contents should be displayed with rich markup and the parts matched by the scanner rules highlighted, with more information provided in tooltips or popovers elements.

The explanation part consists of the three tabs *Description*, *How to fix* and *References*, which are displayed for this specific issue in figures 40, 41 and 42 respectively. The whole view of the issue is modelled after the *Security Hotspot* view in *SonarQube* and the suggestions found in the literature on software security education, as presented in section 2.3.

What we did not see so far, is how the scan is deleted. What happens in the background when the user clicks on the Delete button in the UI is similar to what figure 43 shows. Here,



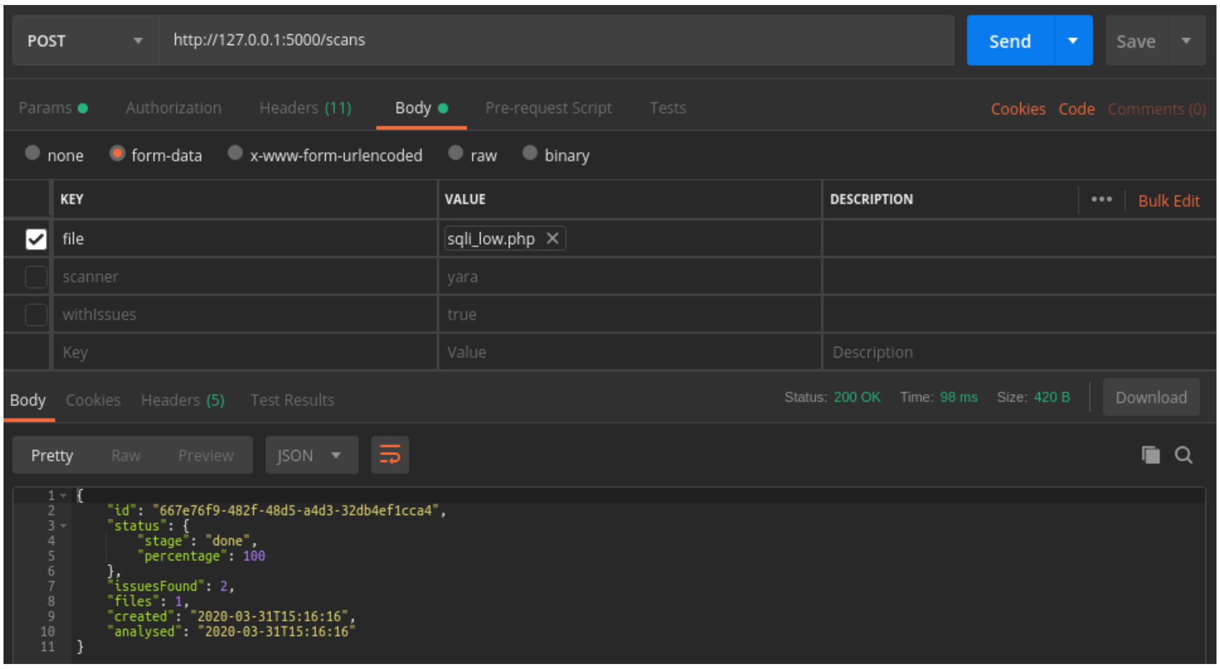
http://127.0.0.1:5000/scans

KEY	VALUE	DESCRIPTION
✓ file	sqli_low.php	
✓ scanner	yara	
✓ withIssues	true	
Key	Value	Description

Status: 200 OK Time: 63 ms Size: 1.94 KB

```
{
    "id": "785676db-675c-4777-91bc-694e3715336b",
    "status": {
        "stage": "done",
        "percentage": 100
    },
    "issuesFound": 1,
    "files": 1,
    "created": "2020-03-31T15:15:30",
    "analysed": "2020-03-31T15:15:30",
    "issues": [
        {
            "source": {
                "scanner": "yara",
                "rule": "SQLi",
                "info": "https://virustotal.github.io/yara/",
                "cli": "yara -r -w -s -m /opt/scrap/scanners/yara/scrap.yar uploads/785676db-675c-4777-91bc-694e3715336b/sqli_low.php"
            },
            "type": "SQLi",
            "explanation": "yara.SQLi",
            "affectedFiles": [
                {
                    "path": "sqli_low.php",
                    "lines": [
                        {
                            "characters": {
                                "from": 65,
                                "to": 89
                            },
                            "text": "b\"$id = $_REQUEST[ 'id' ];\"",
                            "description": "Your code might be vulnerable to an SQL injection | The $id parameter seems to not be sanitized | More
                                info at: https://scrap/description/sqli"
                        },
                        {
                            "characters": {
```

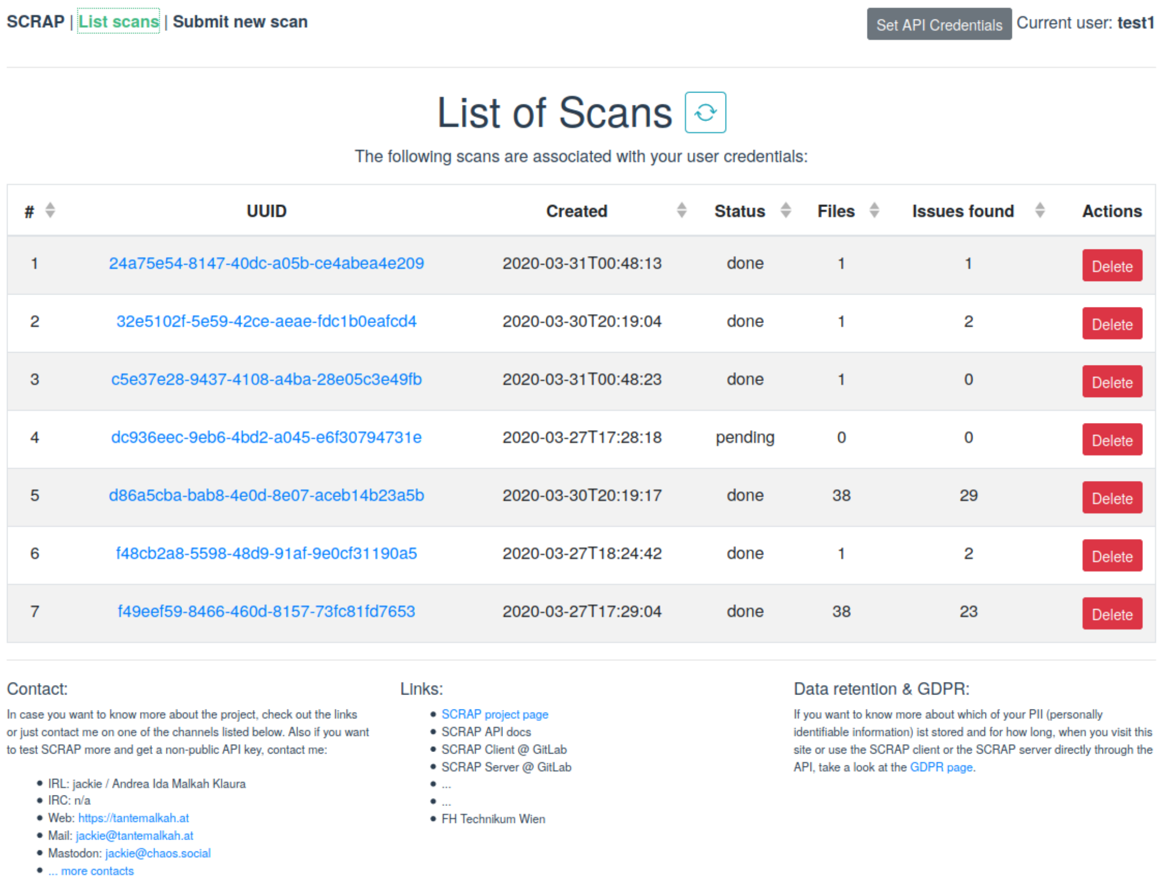
Figure 36: POSTing a new scan with Postman, facilitating only the yara scanner and including the results



	KEY	VALUE	DESCRIPTION
✓	file	sqli_low.php	
	scanner	yara	
	withIssues	true	
	Key	Value	Description

```
{
    "id": "667e76f9-482f-48d5-a4d3-32db4ef1cca4",
    "status": {
        "stage": "done",
        "percentage": 100
    },
    "issuesFound": 2,
    "files": 1,
    "created": "2020-03-31T15:16:16",
    "analysed": "2020-03-31T15:16:16"
}
```

Figure 37: POSTing a new scan with Postman, facilitating all scanners without immediate results



List of Scans

The following scans are associated with your user credentials:

#	UUID	Created	Status	Files	Issues found	Actions
1	24a75e54-8147-40dc-a05b-ce4abea4e209	2020-03-31T00:48:13	done	1	1	Delete
2	32e5102f-5e59-42ce-aeae-fdc1b0eafcd4	2020-03-30T20:19:04	done	1	2	Delete
3	c5e37e28-9437-4108-a4ba-28e05c3e49fb	2020-03-31T00:48:23	done	1	0	Delete
4	dc936eec-9eb6-4bd2-a045-e6f30794731e	2020-03-27T17:28:18	pending	0	0	Delete
5	d86a5cba-bab8-4e0d-8e07-aceb14b23a5b	2020-03-30T20:19:17	done	38	29	Delete
6	f48cb2a8-5598-48d9-91af-9e0cf31190a5	2020-03-27T18:24:42	done	1	2	Delete
7	f49eef59-8466-460d-8157-73fc81fd7653	2020-03-27T17:29:04	done	38	23	Delete

Contact:

In case you want to know more about the project, check out the links or just contact me on one of the channels listed below. Also if you want to test SCRAP more and get a non-public API key, contact me:

- IRL: jackie / Andrea Ida Malkah Klaura
- IRC: n/a
- Web: https://tantemalkah.at
- Mail: jackie@tantemalkah.at
- Mastodon: jackie@chaos.social
- ... more contacts

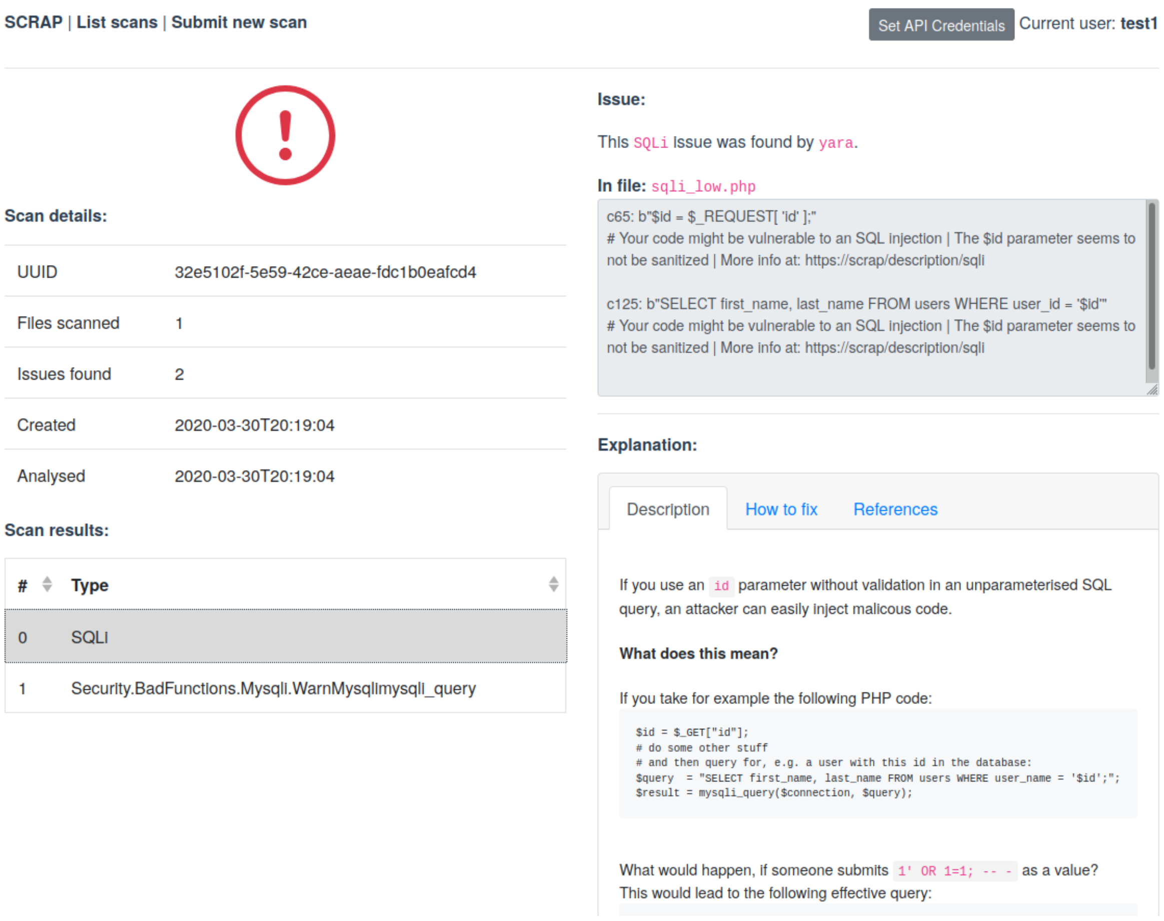
Links:

- SCRAP project page
- SCRAP API docs
- SCRAP Client @ GitLab
- SCRAP Server @ GitLab
- ...
- ...
- FH Technikum Wien

Data retention & GDPR:

If you want to know more about which of your PII (personally identifiable information) ist stored and for how long, when you visit this site or use the SCRAP client or the SCRAP server directly through the API, take a look at the GDPR page.

Figure 38: Listing of all available scans in the SCRAP web UI



SCRAP | List scans | Submit new scan

Set API Credentials Current user: **test1**

Scan details:

UUID	32e5102f-5e59-42ce-aeae-fdc1b0eafcd4
Files scanned	1
Issues found	2
Created	2020-03-30T20:19:04
Analysed	2020-03-30T20:19:04

Scan results:

#	Type
0	SQLi
1	Security.BadFunctions.Mysqli.WarnMysqlimysqli_query

Issue:

This `SQLi` issue was found by `yara`.

In file: `sqli_low.php`

```
c65: b"$id = $_REQUEST[ 'id' ];"
# Your code might be vulnerable to an SQL injection | The $id parameter seems to not be sanitized | More info at: https://scrap/description/sqli

c125: b"SELECT first_name, last_name FROM users WHERE user_id = '$id'"
# Your code might be vulnerable to an SQL injection | The $id parameter seems to not be sanitized | More info at: https://scrap/description/sqli
```

Explanation:

Description | How to fix | References

If you use an `id` parameter without validation in an unparameterised SQL query, an attacker can easily inject malicous code.

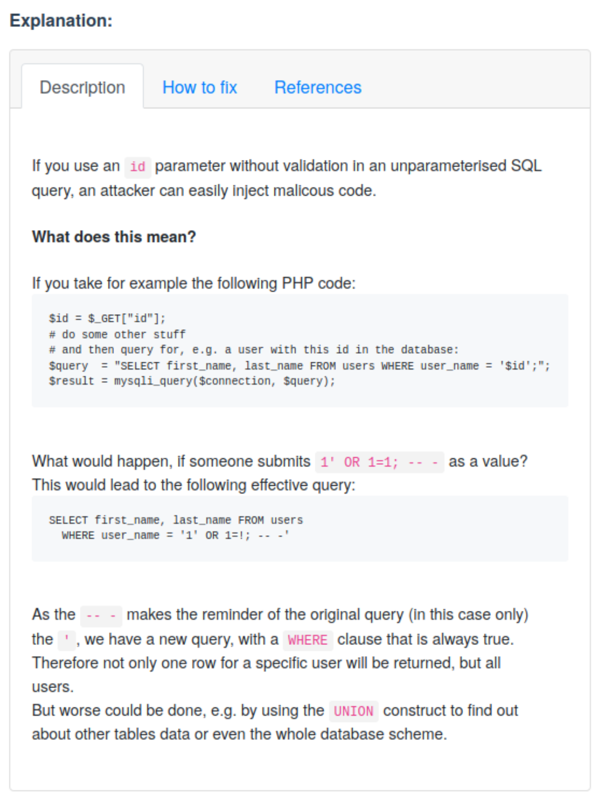
What does this mean?

If you take for example the following PHP code:

```
$id = $_GET["id"];
# do some other stuff
# and then query for, e.g. a user with this id in the database:
$query  = "SELECT first_name, last_name FROM users WHERE user_name = '$id';";
$result = mysqli_query($connection, $query);
```

What would happen, if someone submits `1' OR 1=1; -- -` as a value?
This would lead to the following effective query:

Figure 39: View for one specific scan in the SCRAP web UI



If you use an `id` parameter without validation in an unparameterised SQL query, an attacker can easily inject malicous code.

What does this mean?

If you take for example the following PHP code:

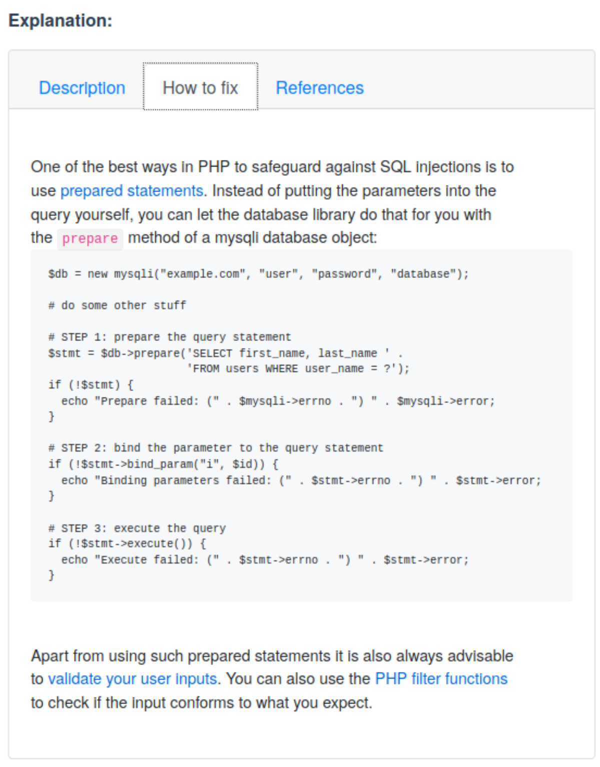
```
$id = $_GET["id"];
# do some other stuff
# and then query for, e.g. a user with this id in the database:
$query  = "SELECT first_name, last_name FROM users WHERE user_name = '$id';";
$result = mysqli_query($connection, $query);
```

What would happen, if someone submits `1' OR 1=1; -- -` as a value?
This would lead to the following effective query:

```
SELECT first_name, last_name FROM users
  WHERE user_name = '1' OR 1=!; -- -'
```

As the `-- -` makes the reminder of the original query (in this case only) the `'`, we have a new query, with a `WHERE` clause that is always true. Therefore not only one row for a specific user will be returned, but all users.
But worse could be done, e.g. by using the `UNION` construct to find out about other tables data or even the whole database scheme.

Figure 40: Description section of an explanation in the SCRAP web UI

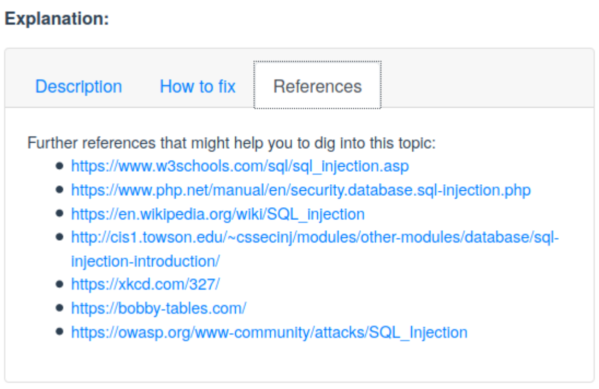


One of the best ways in PHP to safeguard against SQL injections is to use prepared statements. Instead of putting the parameters into the query yourself, you can let the database library do that for you with the `prepare` method of a mysqli database object:

```
$db = new mysqli("example.com", "user", "password", "database");

# do some other stuff

# STEP 1: prepare the query statement
$stmt = $db->prepare('SELECT first_name, last_name ' .
                     'FROM users WHERE user_name = ?');
if (!$stmt) {
  echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

# STEP 2: bind the parameter to the query statement
if (!$stmt->bind_param("i", $id)) {
  echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

# STEP 3: execute the query
if (!$stmt->execute()) {
  echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}
```

Apart from using such prepared statements it is also always advisable to validate your user inputs. You can also use the PHP filter functions to check if the input conforms to what you expect.

Figure 41: How to fix section of an explanation in the SCRAP web UI

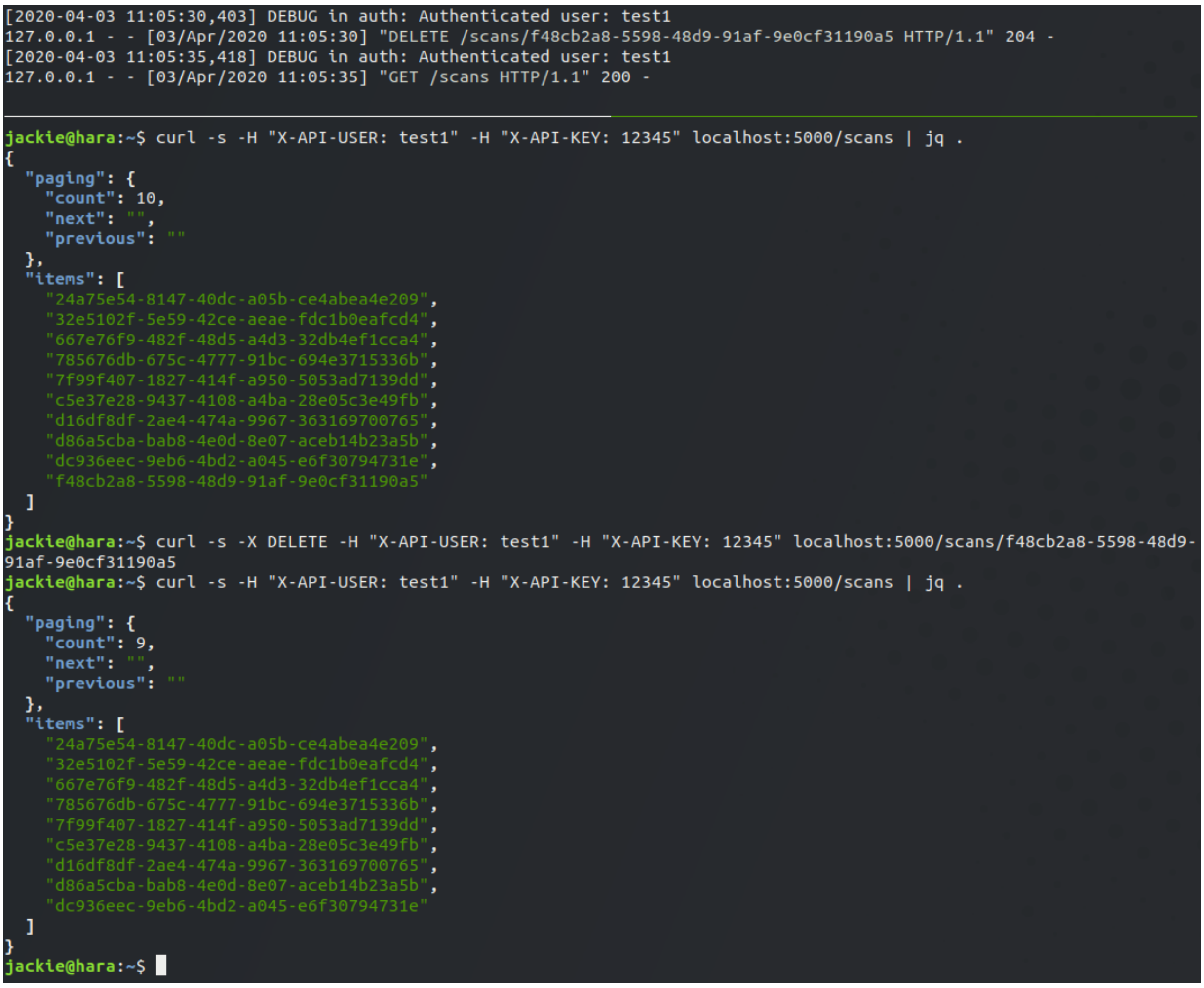

Explanation:

Further references that might help you to dig into this topic:

- https://www.w3schools.com/sql/sql_injection.asp
- https://www.php.net/manual/en/security.database.sql-injection.php
- https://en.wikipedia.org/wiki/SQL_injection
- http://cis1.towson.edu/~cssecinj/modules/other-modules/database/sql-injection-introduction/
- https://xkcd.com/327/
- https://bobby-tables.com/
- https://owasp.org/www-community/attacks/SQL_Injection

Figure 42: References section of an explanation in the SCRAP web UI

after a listing of available scans is requested, a simple DELETE request for a specific scan is sent to the server, before another listing of scans is requested to check if the scan was actually deleted.


```
[2020-04-03 11:05:30,403] DEBUG in auth: Authenticated user: test1
127.0.0.1 - - [03/Apr/2020 11:05:30] "DELETE /scans/f48cb2a8-5598-48d9-91af-9e0cf31190a5 HTTP/1.1" 204 -
[2020-04-03 11:05:35,418] DEBUG in auth: Authenticated user: test1
127.0.0.1 - - [03/Apr/2020 11:05:35] "GET /scans HTTP/1.1" 200 -
```

```
jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans | jq .
{
  "paging": {
    "count": 10,
    "next": "",
    "previous": ""
  },
  "items": [
    "24a75e54-8147-40dc-a05b-ce4abea4e209",
    "32e5102f-5e59-42ce-aeae-fdc1b0eafcd4",
    "667e76f9-482f-48d5-a4d3-32db4ef1cca4",
    "785676db-675c-4777-91bc-694e3715336b",
    "7f99f407-1827-414f-a950-5053ad7139dd",
    "c5e37e28-9437-4108-a4ba-28e05c3e49fb",
    "d16df8df-2ae4-474a-9967-363169700765",
    "d86a5cba-bab8-4e0d-8e07-aceb14b23a5b",
    "dc936eec-9eb6-4bd2-a045-e6f30794731e",
    "f48cb2a8-5598-48d9-91af-9e0cf31190a5"
  ]
}
jackie@hara:~$ curl -s -X DELETE -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans/f48cb2a8-5598-48d9-
91af-9e0cf31190a5
jackie@hara:~$ curl -s -H "X-API-USER: test1" -H "X-API-KEY: 12345" localhost:5000/scans | jq .
{
  "paging": {
    "count": 9,
    "next": "",
    "previous": ""
  },
  "items": [
    "24a75e54-8147-40dc-a05b-ce4abea4e209",
    "32e5102f-5e59-42ce-aeae-fdc1b0eafcd4",
    "667e76f9-482f-48d5-a4d3-32db4ef1cca4",
    "785676db-675c-4777-91bc-694e3715336b",
    "7f99f407-1827-414f-a950-5053ad7139dd",
    "c5e37e28-9437-4108-a4ba-28e05c3e49fb",
    "d16df8df-2ae4-474a-9967-363169700765",
    "d86a5cba-bab8-4e0d-8e07-aceb14b23a5b",
    "dc936eec-9eb6-4bd2-a045-e6f30794731e"
  ]
}
jackie@hara:~$ 
```

Figure 43: Deleting a scan with curl

The prototype functionality that is documented here was developed in about a month of full-time work. The workload for the different components is provided in the following list, ordered in the chronology of the components development:

1. Scanner evaluation: 10 person days (this is not strictly part of the implementation phase, but it was the logical first step to test the 7 scanners in detail, before implementing 2 scanners in the final prototype)
2. API design: 3 person days
3. Server implementation: 10 person days
4. Web UI implementation: 3 person days

All this is based on some prior, but not extensive experience with the Flask and Vue.js frameworks and Python and Javascript skills of a long-time system administrator with part-time developer aspirations. This shows, that the technical side of the SCRAP approach should be reasonably doable for an educational organisation with a computer science faculty. My estimate is, that with another person month the prototype could be developed into a first fully functional production version, which then could be integrated into existing code submission systems and tested with actual students' exercise code. How much the integration effort itself will be depends of course on the architecture of the systems in place.

Much more effort than into the technical implementation will have to go into content creation. On the one side a substantial amount of new scanner rules will be needed, tailored towards the requirements of web programming courses and tested on actual code submissions from prior courses. On the other side also good explanations have to be created for the issues found with those new rules, additionally to explanations for already existing rules which come already on board with the used scanners.

Then again, the approach shows that it would not need a big effort to adopt other scanners, either for PHP or other code. So the whole system would not have to be used for web programming courses but could be used for any introductory programming contexts, which are probably more common. Also the existing scanners for languages like C/C++, Java or Python, which are popular in introductory programming contexts, might provide better out-of-the-box results than the scanners found for PHP code.

Another big question that this research has to leave open for future investigations is, if the adoption of *SonarQube* might provide a better fit, given that the vulnerability scanning for PHP is increased and maybe is already much better for other languages in the current version. This was not in scope for this thesis, but that would be a viable path for other thesis to follow. After all, the display of issues and explanations in the SCRAP UI is already modelled after what *SonarQube* provides in its new *Security Hotspots* view. And while SCRAP is rather lightweight and therefore simple to integrate, *SonarQube* is a full-fledged project with an established user and developer community.

Another issue, that would have to be solved on a socio-technical policy level within the organisation, is that of coding standards. As for example *PHP_CodeSniffer* shows, there is already ample support for checking against different coding standards. And the results are then much more usable. And while adherence to a coding standard might seem to be too much overhead for introductory programming courses, the research on software security education shows that adoption of a secure coding mindset is most efficient, if secure coding is integrated from the early start on. This is in line with research in software security and secure development life cycles in general. The earlier the security perspective is introduced, the better and less prone to vulnerabilities the results are.

So while SCRAP might be partly useful as an add-on to existing introductory programming courses and other educational programming contexts, we know that security as an add-on just does not do what we want it to do. So the main gain of a solution like SCRAP or the adoption of *SonarQube* or other mature platforms will only be unfolded if the organisational context it is deployed in also adopts its policies and didactic strategies. This then will probably be the most effort in adoption of approaches likes SCRAP.

6 Future research

Based on current research in software security education, as described in section 2.3 and the SCRAP prototype, described in chapter 5, there are a few questions end experiments which could further explore the applicability of the SCRAP approach and shed some light on how to improve actual integration of secure coding topics into computer science curricula.

First of all, the current approach focuses on web application development and therefore specifically on PHP. A similar investigation that was conducted here for F/LOSS SAST tools dealing with PHP code could find F/LOSS scanners for C/C++, Java and Python, or other languages which might provide significantly better out-of-the-box results.

Then it would be worthwhile, either with the current state of SCRAP or with additional scanners for other languages included, to evaluate the SCRAP approach with other data sets than the *DVWA* vulnerability files. It is very much likely that some other data sets might result in better coverage of vulnerabilities. Also the results would provide additional examples for more concrete scanner rules that could be developed. The best data set would, of course, be actual code submitted by students in introductory (web) programming courses.

As the evaluation of the static analysers for PHP in section 5.3 showed, *SonarQube* provides a framework that could be used instead of building a new platform like SCRAP, given that its *Security Hotspot* feature gets improved and covers more PHP code vulnerabilities. It therefore might be a valuable first step to investigate the necessary effort to adopt *SonarQube* in order to produce satisfactory results for PHP code. Additionally it should be tested if it already provides better results for other languages.

After SCRAP is improved, based on further research in the areas mentioned so far, its application in a real-life scenario could be tested, based on programming courses at the FHTW. A mix of quantitative evaluation based on online interview plus a few qualitative interviews and situational analysis as applied in my former thesis computer scientists and their publics (Klaura, 2014). The primary questions that this research should answer are 1) how well students understand the generated feedback and 2) if they perceive the generated feedback as an added value to the rest of the course materials.

Finally, an extension of the platform by smaller gamification features could increase student engagement and interactivity, in order to increase retainment of the learned contexts and adoption into daily coding practices.

7 Conclusion

The main contribution of this thesis is twofold:

1. There is no comprehensive overview of the field of software security education, at least not one that is easy to find and accessible. While one such review was found for the area of introductory programming in general (Sorva, 2013), the thesis at hand can now provide a reasonable first overview for the state of the art of software security education. I hope this can help other researchers new to this field. It certainly would have propelled the SCRAP approach a bit further, if I had found such an overview.
2. The SCRAP prototype can be used as a basis for further research in this field. It is designed in a RESTful style and with an API-first approach, that provides a common basis for adoption and extensions with other components, as well as the integration into existing systems. The opportunities for further research, as described in the previous section, are manifold.

An important aspect highlighted by my research is one that is not new to IT security research at all, but still so often neglected: integrating security as an add-on does not provide satisfactory results. What my thesis shows specifically, in line with other researchers in this field, is that this is also true for the issue of secure coding and software security education. Integrating security and secure coding from the start on, into introductory programming courses, could be a main lever to improve overall software security.

One problem that we have in this area, is that software security education is still significantly under-researched, and more efforts have to be put into this field of research not only by individual institutions who want to improve their own students' capabilities to produce reliable and secure software for our increasingly digitally connected world. Rather this has to be a collective effort, and we would do good to foster more sustainable and cooperative research in this area.

Nevertheless, there are several promising results and projects out there, as my literature review also showed. These could be seen as impulses to take up, to connect our research and to work towards more standardised solutions and openly accessible databases for vulnerability scanning and explanations tied to the needs of computer science education and introductory programming contexts. Because one huge problem seems to be that from time to time there a promising project starts, and then, at some point it ends, usually due to the ending of a funding period. Sustainability is hard to achieve when everyone is trying to solve their problem on their own and has to compete for limited funding.

The approach followed in this work, to come up with a new prototype platform called SCRAP, to scan students' submissions to coding exercises for vulnerabilities and to provide feedback how to mitigate those, before this code is usually scrapped without further benefits other than passing an exam, is certainly do-able, also for smaller organisations, as highlighted in more detail in the SCRAP evaluation section 5.3. The main problem does not seem to be a technical one of implementing appropriate toolchains an web services, but of producing useful content and adopting organisational structures and didactic approaches. Future research might just as well find that SCRAP should be scrapped in favour a platform like *SonarQube* - and given a corresponding adoption of coding standards and policies and didactic methods at an higher education institution this might prove very beneficial. The main problem yet stays the same: if an institution wants to adopt this as an add-on to existing (educational and technological) structures, in the hopes that no additional effort has to be taken in other areas, the approach will most likely fail and the efforts to integrate it could have been used better somewhere else.

The SCRAP prototype that was developed is featured on the project websites https://scrap.tantemalkah.at, with a bit of background information to this work and links to the repositories for the prototype RESTful web service, the prototype web UI and the evaluation of the different PHP scanners found. Also the API definition and documentation can be found there. The evaluation section 5.3 describes in more detail the feasibility and effort needed to facilitate such an approach. In short: it can certainly be done without too much effort on the technical side. But, as elaborated above, a solution on the technical level alone will not prove satisfactory.

When it comes specifically to the available F/LOSS tools for scanning PHP code, the thesis found, that on the whole they do not seem to have a good coverage of secure coding issues. A significant effort would have to be made to improve them. This might be different for other languages like C/C++, Java or Python. But the approach taken by SCRAP would certainly lend itself to the adoption of other scanners and programming languages to scan.

Bibliography

Alonso, G., Casati, F., Kuno, H. & Machiraju, V., 2004. *Web Services: Concepts, Architectures and Applications*. Berlin, Heidelberg: Springer Verlag.

Anis, A., Zulkernine, M., Iqbal, S., Liem, C. & Chambers, C., 2018. Securing Web Applications with Secure Coding Practices and Integrity Verification. In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. 2018, Athens, Greece: IEEE, pp.618–625.

@anthonypjshaw, 2018. *10 common security gotchas in Python and how to avoid them*. [Online] Available at: <https://hackernoon.com/10-common-security-gotchas-in-python-and-how-to-avoid-them-e19fbe265e03> [Accessed 2020-04-04].

Bishop, M., 2006. Teaching context in information security. *Journal on Educational Resources in Computing (JERIC)*, 6(3).

Bishop, M., Miloslavskaya, N. & Theocharidou, M. eds., 2015. *Information Security Education Across the Curriculum*. IFIP Advances in Information and Communication Technology. 1st edition. Hamburg, Germany: Springer.

Boulay, B. D., 1986. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), pp.57–73.

Bouwers, E., 2006. *Php Sat Origin*. [Online] Available at: <http://program-transformation.org/PHP/PhpSatOrigin> [Accessed 2020-02-21].

Bruce-Lockhart, M. P. & Norvell, T. S., 2007. Developing Mental Models of Computer Programming Interactively Via the Web. In: *37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports*. 2007, Milwaukee, USA: IEEE, pp.S3H–3–S3H–8.

Buchele, S. F., 2013. Two Models of a Cryptography and Computer Security Class in a Liberal Arts Context. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. 2013, Denver, USA: ACM, pp.543–548.

Bunke, M., 2015. Software-security patterns: Degree of maturity. In: *Proceedings of the 20th European Conference on Pattern Languages of Programs*. 2015, Kaufbeuren, Germany: ACM, pp.42:1–42:17.

Cappos, J. & Weiss, R., 2014. Teaching the Security Mindset with Reference Monitors. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. 2014, Atlanta, USA: ACM, pp.523–528.

CERN Computer Security Team, 2020. *CERN Computer Security Information: Static Code Analysis Tools*. [Online] Available at: <https://security.web.cern.ch/security/recommendations/en/code_tools.shtml> [Accessed 2020-02-21].

Chi, H., Jones, E. L. & Brown, J., 2013. Teaching Secure Coding Practices to STEM Students. In: *Proceedings of the 2013 on InfoSecCD '13: Information Security Curriculum Development Conference*. 2013, Kennesaw, USA: ACM, pp.42:42–42:48.

Code Dx, 2018. *Predicted web application vulnerabilities and cybersecurity trends for 2019*. [Online] Available at: <https://codedx.com/predicted-web-application-vulnerabilities-and-cybersecurity-trends-for-2019/> [Accessed 2019-09-11].

Conklin, W. A., White, G., Williams, D., Davis, R. & Cothren, C., 2016. *Principles of Computer Security*. 4th edition. [eBook] : McGraw-Hill Education. Available at: <https://www.mhprofessional.com/9780071836012-usa-principles-of-computer-security-fourth-edition> [Accessed 2019-07-11].

Dewhurst Security, 2020. *Damn Vulnerable Web Application (DVWA)*. [Online] Available at: <http://www.dvwa.co.uk/> [Accessed 2020-02-13].

docker docs, 2020. *Get Docker Engine - Community for Debian*. [Online] Available at: <https://docs.docker.com/install/linux/docker-ce/debian/> [Accessed 2020-04-01].

Endler, M., 2020. *mre/awesome-static-analysis: Static analysis tools for all programming languages*. [Online] Available at: <https://github.com/mre/awesome-static-analysis> [Accessed 2020-02-21].

FH Technikum Wien, 2019. *Bachelor-Studiengang VZ Informatik Studienplan*. [Online] Available at: <https://www.technikum-wien.at/file/4712/download/> [Accessed 2019-09-13].

Fielding, R. T., 2000. *Architectural Styles and the Design of Network-based Software Architectures*. [Dissertation] University of California, Irvine: University of California, Irvine. Available at: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> [Accessed 2020-04-23].

FIRST, 2019. *Common Vulnerability Scoring System SIG*. [Online] Available at: <https://www.first.org/cvss/> [Accessed 2019-09-18].

Giannakas, F., Kambourakis, G., Papasalouros, A. & Gritzalis, S., 2018. A critical review of 13 years of mobile game-based learning; a bi-monthly publication of the association for educational communications & technology. *Educational Technology Research and Development*, 66(2), pp.341–384.

github.com/phan/phan, 2020. *Incrementally Strengthening Analysis*. [Online] Available at: <https://github.com/phan/phan/wiki/Incrementally-Strengthening-Analysis> [Accessed 2020-04-01].

github.com/squizlabs/PHP_CodeSniffer, 2016. *squizlabs/PHP_CodeSniffer Wiki: Home*. [Online] Available at: <https://github.com/squizlabs/PHP_CodeSniffer/wiki> [Accessed 2020-04-01].

Hadnagy, C., 2011. *Social Engineering: The Art of Human Hacking*. Indianapolis, Indiana: Wiley Publishing, Inc.

Helisch, M. & Pokoyski, D. eds., 2009. *Security Awareness. Neue Wege zur erfolgreichen Mitarbeiter-Sensibilisierung*. Wiesbaden: Vieweg + Teubner.

Heymann, E. & Miller, B., 2018. Tutorial: Secure Coding Practices, Automated Assessment Tools and the SWAMP. In: *2018 IEEE Cybersecurity Development (SecDev)*. 2018, Cambridge, USA: IEEE, pp.124–125.

Hooshangi, S., Weiss, R. & Cappos, J., 2015. Can the Security Mindset Make Students Better Testers?. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 2015, Kansas City, USA: ACM, pp.404–409.

Horster, P., 1985. *Kryptologie*. Mannheim, Wien: BI-Wiss.-Verlag.

Howard, M. & Le Blanc, D., 2003. *Writing secure code: practical strategies and proven techniques for building secure applications in a networked world*. 2nd edition. Redmond: Microsoft Press.

ISO, 2013. *ISO/IEC 27001:2013*. [Online] Available at: <https://www.iso.org/obp/ui/#iso:std:iso-iec:27001:ed-2:v1:en> [Accessed 2019-09-15].

Jawed, M., 2019. *Continuous security in DevOps environment: Integrating automated security checks at each stage of continuous deployment pipeline*. [Master Thesis] Technische Universität Wien. Available at: <https://repositum.tuwien.ac.at/urn:nbn:at:at-ubtuw:1-124776> [Accessed 2020-04-23].

Jøsang, A., Ødegaard, M. & Oftedal, E., 2015. Cybersecurity Through Secure Software Development. In: *Information Security Education Across the Curriculum*. 2015, Hamburg, Germany: Springer, pp.53–63.

Kaza, S. & Taylor, B., 2018. Introducing Secure Coding in Undergraduate (CS0, CS1, and CS2) and High School (AP Computer Science A) Programming Courses (Abstract Only). In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. PT2018, Baltimore, USA: ACM, pp.1050–1050.

Kaza, S., Taylor, B. & Hawthorne, E. K., 2015. Introducing Secure Coding in CS0, CS1, and CS2: Conference Workshop. *J. Comput. Sci. Coll.*, 30(6), pp.11–12.

Kaza, S., Taylor, B. & Sherbert, K., 2018. Hello, World!—Code Responsibly. *IEEE Security Privacy*, 16(1), pp.98–100.

Kernegger, K., 2013. *Improving error detection rate using retesting in automated security testing tools*. [Master Thesis] Technische Universität Wien. Available at: <http://repositum.tuwien.ac.at/urn:nbn:at:at-ubtuw:1-68534> [Accessed 2020-04-23].

Klaura, A. I. M., 2014. *Computer scientists & their publics : on constructions of "participation" and "publics" in participatory design and research*. [Master Thesis] Universität Wien. Available at: <https://ubdata.univie.ac.at/AC12140416> [Accessed 2020-04-05].

Kriha, W. & Schmitz, R., 2008. *Internet-Security aus Software-Sicht : Ein Leitfaden zur Software-Erstellung für sicherheitskritische Bereiche*. Berlin: Springer.

Lange, K., 2016. *The Little Book on REST Services*. Copenhagen. Available at: <https://www.kennethlange.com/books/download.php?file=The-Little-Book-on-REST-Services.pdf> [Accessed 2020-04-03].

Lavieri, E. & Verhas, P., 2017. *Mastering Java 9 : Write reactive, modular, concurrent, and secure code*. Birmingham, Mumbai: Packt.

Lindmaier, C., 2016. *Automatisierte Security Tests für Webapplikationen*. [Master Thesis] Fachhochschule Technikum Wien. Available at: <http://permalink.obvsg.at/ftw/AC13291310> [Accessed 2020-02-12].

Meng, N., Nagy, S., Yao, D., Zhuang, W. & Arango-Argoty, G., 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, Gothenburg, Sweden: ACM, pp.372–383.

Miller, D., Whitlock, J., Gardiner, M., Ralphson, M., Ratovsky, R. & Sarid, U., 2020. *OpenAPI Specification Version 3.0.0*. [Online] Available at: <http://spec.openapis.org/oas/v3.0.0> [Accessed 2020-04-02].

Mirtes, O., 2016. *PHPStan: Find Bugs In Your Code Without Writing Tests!*. [Online] Available at: <https://medium.com/@ondrejmirtes/phpstan-2939cd0ad0e3> [Accessed 2020-04-01].

Mirtes, O., 2018*a*. *Looking for Sponsors of the Next Major PHPStan Release!*. [Online] Available at: <https://medium.com/@ondrejmirtes/

looking-for-sponsors-of-the-next-major-phpstan-release-73204cce0666> [Accessed 2020-04-01].

Mirtes, O., 2018*b*. *Next Chapter in PHPStan Saga*. [Online] Available at: <https://medium.com/@ondrejmirtes/next-chapter-in-phpstan-saga-3bfd7ffdb81d> [Accessed 2020-04-01].

MITRE, 2019. *2019 CWE Top 25 Most Dangerous Software Errors*. [Online] Available at: <https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html> [Accessed 2019-09-21].

MITRE & SANS, 2011. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. [Online] Available at: <https://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.html> [Accessed 2019-09-21].

Morgridge Institute for Research, 2020. *Software Assurance Marketplace: Tools*. [Online] Available at: <https://www.mir-swamp.org/#tools/public> [Accessed 2020-02-21].

National Institute of Standards and Technology, 2020. *Source Code Security Analyzers*. [Online] Available at: <https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html> [Accessed 2020-02-21].

Nembhard, F., Carvalho, M. & Eskridge, T., 2019. Towards the application of recommender systems to secure coding. *Eurasip Journal on Information Security*, 2019(9).

OWASP, 2010. *OWASP Secure Coding Practices Quick Reference Guide*. [Online] Available at: <https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf> [Accessed 2020-02-12].

OWASP, 2016. *OWASP ASIDE Project*. [Online] Available at: <https://www.owasp.org/index.php/OWASP_ASIDE_Project> [Accessed 2019-09-06].

OWASP, 2017. *OWASP Top 10 - 2017. The Ten Most Critical Web Application Security Risks*. [Online] Available at: <https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf> [Accessed 2019-09-13].

OWASP, 2019*a*. *Buffer Overflow*. [Online] Available at: <https://www.owasp.org/index.php/Buffer_Overflow> [Accessed 2019-09-16].

OWASP, 2019*b*. *OWASP Top Ten Project*. [Online] Available at: <https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project> [Accessed 2019-09-21].

OWASP, 2019*c*. *Session Management Cheat Sheet*. [Online] Available at: <https://www.owasp.org/index.php/Session_Management_Cheat_Sheet> [Accessed 2019-09-15].

OWASP, 2020*a*. *OWASP Application Security Verification Standard*. [Online] Available at: <https://owasp.org/www-project-application-security-verification-standard/> [Accessed 2020-02-12].

OWASP, 2020*b*. *OWASP Cornucopia*. [Online] Available at: <https://owasp.org/www-project-cornucopia/> [Accessed 2020-02-12].

OWASP, 2020*c*. *OWASP Mutillidae 2 Project*. [Online] Available at: <https://wiki.owasp.org/index.php/OWASP_Mutillidae_2_Project> [Accessed 2020-02-13].

OWASP, 2020*d*. *OWASP Snakes And Ladders*. [Online] Available at: <https://owasp.org/www-project-snakes-and-ladders/> [Accessed 2020-02-12].

OWASP, 2020*e*. *Source Code Analysis Tools*. [Online] Available at: <https://owasp.org/www-community/Source_Code_Analysis_Tools> [Accessed 2020-02-21].

Pancho-Festin, S. & Mendoza, M. J., 2014. Integrating computer security into the undergraduate software engineering classes: Lessons learned. In: *2014 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*. 2014, Wellington, New Zealand: IEEE, pp.395–397.

Pawlowski, S. & Jung, Y., 2015. Social representations of cybersecurity by university students and implications for instructional design. *Journal of Information Systems Education*, 26(4), pp.281–294.

php.net, 2019*a*. *PHP 5 ChangeLog*. [Online] Available at: <https://www.php.net/ChangeLog-5.php> [Accessed 2019-09-16].

php.net, 2019*b*. *PHP 7 ChangeLog*. [Online] Available at: <https://www.php.net/ChangeLog-7.php> [Accessed 2019-09-16].

Pompon, R., 2018. *Application Protection Report*. [Online] Available at: <https://www.f5.com/labs/articles/threat-intelligence/2018-Application-Protection-Report> [Accessed 2019-09-11].

Popa, M., 2012. Requirements of a better secure program coding. *Informatica Economica*, 16(4), pp.93–104.

Positive Technologies, 2018. *Web Application Vulnerabilities Statistics for 2017*. [Online] Available at: <https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Web-application-vulnerabilities-2018-eng.pdf> [Accessed 2019-09-11].

Positive Technologies, 2019*a*. *Penetration testing of corporate information systems: statistics and findings, 2019*. [Online] Available at: <https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/> [Accessed 2019-09-11].

Positive Technologies, 2019*b*. *Web application vulnerabilities: statistics for 2018*. [Online] Available at: <https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/> [Accessed 2019-09-11].

Pournaghshband, V., 2013. Teaching the Security Mindset to CS1 Students. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. 2013, Denver, USA: ACM, pp.347–352.

Rahaman, S., Meng, N. & Yao, D., 2018. Tutorial: Principles and Practices of Secure Crypto Coding in Java. In: *2018 IEEE Cybersecurity Development (SecDev)*. 2018, Cambridge, USA: IEEE, pp.122–123.

Raina, S., Kaza, S. & Taylor, B., 2016. Security Injections 2.0: Increasing Ability to Apply Secure Coding Knowledge Using Segmented and Interactive Modules in CS0. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 2016, Memphis, USA: ACM, pp.144–149.

Raina, S., Taylor, B. & Kaza, S., 2014. Interactive e-Learning Modules for Teaching Secure: A Pilot Study (Abstract Only). In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. 2014, Atlanta, USA: ACM, pp.719–720.

Raina, S., Taylor, B. & Kaza, S., 2015. Security Injections 2.0: Increasing Engagement and Faculty Adoption Using Enhanced Secure Coding Modules for Lower-Level Programming Courses. In: *Information Security Education Across the Curriculum*. 2015, Hamburg, Germany: Springer, pp.64–74.

restfulapi.net, 2020. *REST API Tutorial*. [Online] Available at: <https://restfulapi.net/> [Accessed 2020-04-03].

Rohr, M., 2018. *Sicherheit von Webanwendungen in der Praxis : Wie sich Unternehmen schützen können – Hintergründe, Maßnahmen, Prüfverfahren und Prozesse*. 2nd edition. Wiesbaden: Springer Vieweg.

Sahu, D. & Tomar, D., 2017. Analysis of Web Application Code Vulnerabilities using Secure Coding Standards. *Arabian Journal for Science and Engineering*, 42(2), pp.885–895.

Schuckert, F., Katt, B. & Langweg, H., 2017. Source Code Patterns of SQL Injection Vulnerabilities. In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 2017, Reggio Calabria, Italy: ACM, pp.72:1–72:7.

Snyder, C., Myer, T. & Southwell, M., 2010. *Pro PHP Security : from Application Security Principles to the Implementation of XSS Defenses*. 2nd edition. New York: Apress.

SonarQube, 2020. *Get Started in Two Minutes Guide*. [Online] Available at: <https://docs.sonarqube.org/latest/setup/get-started-2-minutes/> [Accessed 2020-04-01].

Sorva, J., 2012. *Visual program simulation in introductory programming education; Visuaalinen ohjelmasimulaatio ohjelmoinnin alkeisopetuksessa*. Aalto University publication series DOCTORAL DISSERTATIONS; 61/2012. [Dissertation] Aalto, Finland: Aalto University; Aalto-yliopisto. Available at: <http://urn.fi/URN:ISBN:978-952-60-4626-6> [Accessed 2020-04-23].

Sorva, J., 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE)*, 13(2), pp.1–31.

Sorva, J., Karavirta, V. & Malmi, L., 2013. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.*, 13(4).

Stallings, W. & Brown, L., 2015. *Computer security : principles and practice*. 3rd edition. Boston: Pearson.

Stallings, W. & Brown, L., 2018. *Computer Security: Principles and Practice*. 4th global edition. New York: Pearson Education.

Stanford Report, 2012. *Stanford launches Class2Go, an open-source platform for online classes*. [Online] Available at: <https://news.stanford.edu/news/2012/september/class2go-online-platform-091212.html> [Accessed 2019-09-23].

Stivalet, B. & Fong, E., 2016. Large Scale Generation of Complex and Faulty PHP Test Cases. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016, Chicago: IEEE, pp.409–415.

swagger.io, 2020. *Best Practices in API Design*. [Online] Available at: <https://swagger.io/resources/articles/best-practices-in-api-design/> [Accessed 2020-04-02].

Taylor, B. & Kaza, S., 2016. Security Injections@Towson: Integrating Secure Coding into Introductory Computer Science Courses. *Trans. Comput. Educ.*, 16(4), pp.16:1–16:20.

Taylor, B., Bishop, M., Hawthorne, E. & Nance, K., 2013. Teaching Secure Coding: The Myths and the Realities. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. 2013, Denver, USA: ACM, pp.281–282.

Technische Universität Wien, 2019*a*. *Studienplan (Curriculum) für das Bachelorstudium Medieninformatik und Visual Computing E 033 532*. [Online] Available at: <https://informatics.tuwien.ac.at/bachelor-ue033532.pdf> [Accessed 2019-09-13].

Technische Universität Wien, 2019*b*. *Studienplan (Curriculum) für das Bachelorstudium Software & Information Engineering E 033 534*. [Online] Available at: <https://informatics.tuwien.ac.at/bachelor-ue033534.pdf> [Accessed 2019-09-13].

Teto, J. K., Bearden, R. & Lo, D. C.-T., 2017. The Impact of Defensive Programming on I/O Cybersecurity Attacks. In: *Proceedings of the SouthEast Conference*. 2017, Kennesaw, USA: ACM, pp.102–111.

Thalheimer, W., 2008. *Providing Learners with Feedback—Part 1: Research-based recommendations for training, education, and e-learning*. [Online] Available at: <https://www.worklearning.com/wp-content/uploads/2017/10/Providing_Learners_with_Feedback_Part1_May2008.pdf> [Accessed 2019-09-04].

Theisen, C., Williams, L., Oliver, K. & Murphy-Hill, E., 2016. Software Security Education at Scale. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 2016, Austin, USA: IEEE, pp.346–355.

Towson University, 2017. *Cybersecurity Modules: Security Injections|Cyber4All @Towson: Workshops & Talks*. [Online] Available at: <http://cis1.towson.edu/~cssecinj/secure-coding-workshop/workshops/> [Accessed 2020-02-18].

Uskov, A. V., 2013*a*. Hands-On Teaching of Software and Web Applications Security. In: *2013 3rd Interdisciplinary Engineering Design Education Conference*. 2013, Santa Clara, USA: IEEE, pp.71–78.

Uskov, A. V., 2013*b*. Software and Web applications security: state-of-the-art courseware and learning paradigm. In: *2013 IEEE Global Engineering Education Conference (EDUCON)*. 2013, Berlin, Germany: IEEE, pp.608–611.

van der Kleij, F. M., Eggen, T. J., Timmers, C. F. & Veldkamp, B. P., 2012. Effects of feedback in a computer-based assessment for learning. *Computers & Education*, 58(1), pp.263 – 272.

van Niekerk, J. & Futcher, L., 2015. The Use of Software Design Patterns to Teach Secure Software Design: An Integrated Approach. In: *Information Security Education Across the Curriculum*. 2015, Hamburg, Germany: Springer, pp.75–83.

Veracode, 2018. *State of Software Security*. [Online] Available at: <https://www.veracode.com/state-of-software-security-report> [Accessed 2019-09-11].

W3C Working Group, 2004. *Web Services Architecture*. [Online] Available at: <https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest> [Accessed 2020-04-03].

W3Techs, 2020. *Usage statistics of server-side programming languages for websites*. [Online] Available at: <https://w3techs.com/technologies/overview/programming_language> [Accessed 2020-03-03].

Walkinshaw, N., 2017. *Software Quality Assurance; Consistency in the Face of Complexity and Change*. Undergraduate Topics in Computer Science. Cham: Springer International Publishing.

Weir, C., Rashid, A. & Noble, J., 2016. Reaching the Masses: A New Subdiscipline of App Programmer Education. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, Seattle, USA: ACM, pp.936–939.

Whitney, M., Lipford-Richter, H., Chu, B. & Zhu, J., 2015. Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 2015, Kansas City, USA: ACM, pp.60–65.

Wikipedia (EN), 2020. *List of tools for static code analysis*. [Online] Available at: <https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis> [Accessed 2020-02-21].

Williams, Laurie, 2016. *CSC515 Software Security*. [Online] Available at: <https://sites.google.com/a/ncsu.edu/csc515-software-security/> [Accessed 2020-02-18].

Xie, T., Bishop, J., Tillmann, N. & de Halleux, J., 2015. Gamifying Software Security Education and Training via Secure Coding Duels in Code Hunt. In: *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. 2015, Urbana, USA: ACM, pp.26:1–26:2.

Zhu, J., Lipford, H. R. & Chu, B., 2013. Interactive Support for Secure Programming Education. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. 2013, Denver, USA: ACM, pp.687–692.

List of Figures

List of Tables

List of Code

List of Abbreviations

ACM	Association for Computing Machinery
API	Application Programming Interface
CDN	Content delivery network
CD	Continuous delivery
CLI	Command-line interface
CS	Computer Science
CSRF	Cross-site request forgery
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
CWSS	Common Weakness Scoring System
DCOM	Distributed Component Object Model
DREAD	Damage, Reproducibility, Exploitability, Affected users, Discoverability
DVWA	Damn Vulnerable Web Application
ECTS	European Credit Transfer and Accumulation System
FHTW	Fachhochschule Technikum Wien
F/LOSS	Free/Libre and Open Source Software
HTTP	Hypertext Transfer Protocol
ICT	Information & communication technologies
IDE	Integrated development environment
IEEE	Institute of Electrical and Electronics Engineers
IVM	Integrity verification module

JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
MOOC	Massive open online course
MVC	Model–view–controller
NVD	National Vulnerability Database
OS	Operating system
OWASP	Open Web Application Security Project
PMF	PHP Malware Finder
REST	Representational state transfer
SAST	Static application security testing
SCRAP	Secure Code Review Automated Platform
SDLC	Software development life cylce
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SQLi	SQL injection
STEM	Science, technology, engineering, and mathematics
STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege
SWAMP	Software Assurance Marketplace
TODO	*By the mighty witchcraftry of the mother of time!* This part is not yet implemented.
TU	Technische Universität Wien
UI	User interface
URI	Uniform Resource Identifier
UV	University of Vienna
XML	Extensible Markup Language
XSS	Cross-site scripting

8 Appendix A: Helper scripts

analyse.sh:

```
#!/bin/bash

source $(dirname $0)/analyse.inc
BASEURL="http://scrap:9000"
SCANNER="${HOME}/scanners/sonar-scanner/bin/sonar-scanner"
TIMEOUT=10

## This script is for evaluation purposes only!
## Therefore no input checking is done. Make sure to use
## a project (& token) name that does not exist yet.
project=$1
src=$2

log () {
   echo "$(date): $1" 1>&2
}
logempty () {
   echo 1>&2
}

# Right after analysis, the results may not yet be available, because
# the SonarQube server is still analyzing. We can use this function later
# to check if the analysis results are available.
getAnalysisCount () {
   curl -s -u ${USERTOKEN} \
      ${BASEURL}/api/project_analyses/search?project=${project} \
      | jq .paging.total
}

log "Creating project ${project}:"
curl -s -u ${USERTOKEN} \
   -F name=${project} \
   -F project=${project} \
   ${BASEURL}/api/projects/create \
   1>&2
logempty

log "Creating scanning token for ${project}"
scantoken=$(curl -s -u ${USERTOKEN} \
   -F name=${project} \
   ${BASEURL}/api/user_tokens/generate \
  | jq .token | tr -d '"')
log "Token: ${scantoken}"

log "Starting scanner"
${SCANNER} -Dsonar.projectKey=${project} \
  -Dsonar.host.url=${BASEURL} \
  -Dsonar.login=${scantoken} \
  -Dsonar.sources=${src} \
  1>&2


# We have to wait some time until the SonarQube server finished
# the analysis and all results are available, before we can
# actually fetch the results.
aCount=$(getAnalysisCount)
timer=0
while [ ${aCount} -eq 0 -a ${timer} -le ${TIMEOUT} ]; do
  log "Analysis not done yet. Waiting a second."
  ((timer=$timer+1))
  if [ ${timer} -ge ${TIMEOUT} ]; then
    echo "Timeout reached! Aborting analysis."
    exit 1
  fi
  sleep 1
  aCount=$(getAnalysisCount)
done

log "Fetching analysis results"
curl -s -u ${USERTOKEN} \
  ${BASEURL}/api/issues/search?componentKeys=${project}

log "Revoking temporary scan token ${scantoken}"
curl -s -u ${USERTOKEN} \
  -F name=${project} \
  ${BASEURL}/api/user_tokens/revoke \
  1>&2

log "Removing project ${project}"
curl -s -u ${USERTOKEN} \
  -F project=${project} \
  ${BASEURL}/api/projects/delete \
  | jq . \
  1>&2
```

Code 37: Listings/analyse.sh

makeplots.py:

```
#!/usr/bin/python3

import pandas as pd
import matplotlib.pyplot as plt

csv = pd.read_csv('results.csv', sep=';')
full = csv.iloc[:, 0:5] # we don't need the comments column

# to retain the index order as found in the CSV (which was the order of
    evaluation)
index = [
    'graudit',
    'phan',
    'PMF',
    'PHPMD',
    'PHPStan',
    'PHP_CodeSniffer',
    'SonarQube',
]

# columns: scanner, direct hits, hits as unspecific, unspecific
overall = full.iloc[:, [0,2,3,4]]
overall = overall.groupby(['scanner name']).sum().reindex(index)
overall.plot.bar(
    title='Comparison of overall scanner findings',
    rot=5,
    figsize=(10,7),
)
plt.yscale('log')

sqli = full[full['test data set'].str.match('^sqli$')].iloc[:, [0,2,3,4]]
sqli.set_index('scanner name', inplace=True)
sqli.plot.bar(
    title='Comparison of findings for sqli folder',
    rot=10,
    figsize=(7,6),
)

sqli_blind = full[full['test data set'].str.match('^sqli_blind$')].iloc[:,
    [0,2,3,4]]
sqli_blind.set_index('scanner name', inplace=True)
sqli_blind.plot.bar(
    title='Comparison of findinds for sqli_blind folder',
    rot=10,
    figsize=(7,6),
)

xss_r = full[full['test data set'].str.match('^xss_r$')].iloc[:, [0,2,3,4]]
xss_r.set_index('scanner name', inplace=True)
xss_r.plot.bar(
    title='Comparison of findings for xss_r folder',
    rot=10,
    figsize=(7,6),
)

xss_s = full[full['test data set'].str.match('^xss_s$')].iloc[:, [0,2,3,4]]
xss_s.set_index('scanner name', inplace=True)
xss_s.plot.bar(
    title='Comparison of findings for xss_s folder',
    rot=10,
    figsize=(7,6),
)

# show all plots
plt.show()
```

Code 38: Listings/makeplots.py

vuln-data-copy.sh:

```
#!/bin/bash

SOURCEDIRS="sqli sqli_blind xss_d xss_r xss_s"
SOURCESINGLES="\
   sqli/source/low.php \
   sqli/source/medium.php \
   sqli/source/high.php \
   xss_r/source/low.php \
   xss_r/source/medium.php \
   xss_r/source/high.php \
   "

usage () {
   echo "Usage: $0 <DVWA-root> <target-dir>"
   echo
   echo "   DVWA-root   ... root directory of the DVWA source"
   echo "   target-dir  ... directory to copy vulnerability test data to"
   echo "                    (target-dir should not exist)"
   echo
}

if [ "$1" = "-h" -o "$1" = "--help" -o "$1" = "help" ]; then
   usage
   exit
fi

if [ ! -d "$1" ]; then
   echo "First parameter has to be the directory containing DVWA!"
   usage
   exit 1
fi

if [ -z "$2" ]; then
   echo "Second parameter has to be the target dir for vulnerability data"
```

```
    usage
    exit 1
fi

if [ -e "$2" ]; then
    if [ -d "$2" ]; then
        if [ -z "$(ls -A "$2")" ]; then
            echo "Target dir exists and empty. All fine."
        else
            echo "The target directory already exists and has content. Aborting."
            exit 1
        fi
    else
        echo "$2 already exists and is not a directory. Aborting."
        exit 1
    fi
fi

for f in $SOURCEDIRS; do
    if [ ! -d "$1/vulnerabilities/$f" ]; then
        echo "The directory $1/vulnerabilities/$f does not exist!"
        exit 1
    fi
done

for f in $SOURCESINGLES; do
    if [ ! -f "$1/vulnerabilities/$f" ]; then
        echo "The file $1/vulnerabilities/$f does not exist!"
        exit 1
    fi
done

echo "All necessary files and directories seem to be there."
echo "Creating target folders and copying vulnerability test data."

if [ ! -d "$2" ]; then
    mkdir -p "$2"
fi

for f in $SOURCEDIRS; do
    echo "'$1/vulnerabilities/$f' -> '$2/$f'"
    cp -a "$1/vulnerabilities/$f" "$2/$f"
done

for f in $SOURCESINGLES; do
    type=$(echo $f | cut -f 1 -d '/')
    level=$(echo $f | cut -f 3 -d '/')
    cp -av "$1/vulnerabilities/$f" "$2/${type}_${level}"
done

```

```
echo "DONE. Vulnerability test data is now available at ${PWD}/$2"
```

Code 39: Listings/vuln–data–copy.sh

9 Appendix B: SCRAP API definition

```yaml
openapi: 3.0.0

info:
  version: "1.0.0"
  title: SCRAP - Secure Code Review Automated Platform
  description: >-
    SCRAP is a prototype for analysing code submissions by students in regards
    to secure coding and to provide feedback. This API is the main interface
    to the SCRAP server. Details on the whole project can be found on
    [scrap.tantemalkah.at](https://scrap.tantemalkah.at).

servers:
  - description: SCRAP evaluation server (rate limited for unauthenticated
      accounts)
    url: https://scrap.tantemalkah.at/api/v1
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/tantemalkah/SCRAP/1.0.0

tags:
  - name: auth
    description: Only for authenticated users. The server might provide a
        ‘public‘ user.
    externalDocs:
      url: https://scrap.tantemalkah.at/docs/authentication.html
  - name: public
    description: Accessible without API key or other auth method.

paths:
  /:
    get:
      summary: Retrieve the server and API meta information
      description: >-
        This endpoint serves as a sort of welcome page, providing some meta
        information about the server and the API.
      tags:
        - public
      responses:
        ’200’:
          description: Successful transfer of the server’s meta info object
          content:
            application/json:
              schema:
```

```
              $ref: '#/components/schemas/Meta'

/scans:
  get:
    summary: List all available scans of a user
    description: >-
      Retrieve a list of your scans.<br><br>

      **Authentication**: This endpoint is only available with a valid API key.
      The server might provide a `public` API key, which can be used for public
      testing.
    security:
    - ApiKeyAuth: []
      ApiUser: []
    tags:
      - auth
    responses:
      '200':
        description: Successful transfer of a list of scans.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ListOfScans'
      '400':
        $ref: '#/components/responses/BadRequest'
      '401':
        $ref: '#/components/responses/Unauthorized'
      '404':
        $ref: '#/components/responses/NotFound'

  post:
    summary: Submit a new scan
    description: >-
      This operation is used to submit a new scan. This will usually be either
      a single PHP file or a gzipped tar archive containing at least one PHP
          file.
      Future version may adopt other scanners and languages. Consult the
          documentation
      of the server you POST to, which file types are acceptable. Version
          1.0.0 is designed
      for use with PHP only.<br><br>

      **Authentication**: This endpoint is only available with a valid API key.
      The server might provide a _public_ API key, which can be used for public
      testing. If you use the `public` as an API key, be aware, that your scan
      will be visible to every other public user. But the server will usually
          do
      regular cleanups of public scans. Consult your server's documentation on
      if and how often those cleanups are done.
    security:
```

```yaml
      - ApiKeyAuth: []
        ApiUser: []
      tags:
        - auth
      requestBody:
        description: >-
          If you POST to this endpoint, you have to submit at least a file
          as part of a multipart/form-data body. Additionally you can use other
          properties, described below, to customize the scan.
        required: true
        content:
          multipart/form-data:
            schema:
              type: object
              properties:
                scanner:
                  type: string
                  description: >-
                    Use this to scan only with one of the available scanners.
                    If you omit this parameter, all deployed scanners will be used.
                withIssues:
                  type: boolean
                  description: >-
                    Set this to true, if the server should wait for the scan to finish and
                    include a list of all issues found in the 'issues' property of the response.
                file:
                  type: string
                  format: binary
                  description: >-
                    The file or archive you want to be scanned. This should either
                    be a single PHP file ('php'), or a gzipped tar archive ('.tgz' or '.tar.gz').
                    Other files will not be accepted.
              required:
                - file
      responses:
        '200':
          description: >-
            Sucessful submission of a new scan. The new scan object
            is returned in the response body.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Scan'
        '400':
          $ref: '#/components/responses/BadRequest'
        '401':
          $ref: '#/components/responses/Unauthorized'
        '404':
          $ref: '#/components/responses/NotFound'
        '413':
          $ref: '#/components/responses/FileTooBig'
        '415':
          $ref: '#/components/responses/WrongFileType'

  /scans/{id}:
    get:
      summary: Retrieve meta information for a single scan
      description: >-
        This operation provides the meta information for a single scan. This
        includes the scan's current stage and progress, the number of found
        issues and files in the uploaded file/package and the timestamps when
        the scan was created (right after the upload completed) and the analysis
        was completed.
      security:
        - ApiKeyAuth: []
          ApiUser: []
      tags:
        - auth
      parameters:
        - $ref: '#/components/parameters/scanIdParam'
      responses:
        '200':
          description: Sucessfully returned a scan object
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Scan'
        '400':
          $ref: '#/components/responses/BadRequest'
        '401':
          $ref: '#/components/responses/Unauthorized'
        '404':
          $ref: '#/components/responses/NotFound'

    delete:
      summary: Delete a single scan
      description: >-
        Delete one of your scans.<br><br>

        If the server provides a `public` API key, it might prohibit the deletion
        of such _public_ scans and only delete them based on a regular interval.
      security:
        - ApiKeyAuth: []
          ApiUser: []
      tags:
        - auth
      parameters:
        - $ref: '#/components/parameters/scanIdParam'
      responses:
        '204':
          description: Sucessfully deleted the scan.
        '400':
          $ref: '#/components/responses/BadRequest'
        '401':
          $ref: '#/components/responses/Unauthorized'
        '404':
          $ref: '#/components/responses/NotFound'

  /scans/{id}/files:
    get:
      summary: Receive listing of all files of a scan
      description: >-
        This operation returns all files that are part of a scan, that is, the
        one file if a single PHP file was uploaded or all files from the
        uploaded .tgz archive.
      security:
        - ApiKeyAuth: []
          ApiUser: []
      tags:
        - auth
      parameters:
        - $ref: '#/components/parameters/scanIdParam'
      responses:
        '200':
          description: Successful transfer of a list of files
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ListOfFiles'
        '400':
          $ref: '#/components/responses/BadRequest'
        '401':
          $ref: '#/components/responses/Unauthorized'
        '404':
          $ref: '#/components/responses/NotFound'

  /scans/{id}/files/{filepath}:
    get:
      summary: Retrieve a single file from a scan
      description: >-
        Returns the meta information of an uploaded file. The file itself
        can be retrieved through its _blob_ endpoint, which is part of the
        returned meta information.
      security:
        - ApiKeyAuth: []
```

```
          ApiUser: []
      tags:
        - auth
      parameters:
        - $ref: '#/components/parameters/scanIdParam'
        - $ref: '#/components/parameters/filePathParam'
      responses:
        '200':
          description: Successful transfer of the file object
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/File'
        '400':
          $ref: '#/components/responses/BadRequest'
        '401':
          $ref: '#/components/responses/Unauthorized'
        '404':
          $ref: '#/components/responses/NotFound'

  /scans/{id}/blob/{filepath}:
    get:
      summary: Receive a single file from a scan
      description: >-
        Returns the file as it was uploaded (or extracted from the uploaded
            archive)

        The **media type** of the response
        depends on the file, but in most cases it will be _application/x-php_,
            especially
        when the scan consists of a single file. If a whole project was uploaded,
        it could be the case that other files than PHP files will be included in
        an issue, depending on which scanners are available and how they are
            configured.
      security:
        - ApiKeyAuth: []
          ApiUser: []
      tags:
        - auth
      parameters:
        - $ref: '#/components/parameters/scanIdParam'
        - $ref: '#/components/parameters/filePathParam'
      responses:
        '200':
          description: Successful transfer of the file.
          content:
            application/x-php:
              schema:
                type: string
            text/html:
```

```
            schema:
              type: string
          text/css:
            schema:
              type: string
          application/javascript:
            schema:
              type: string
          text/markdown:
            schema:
              type: string
          text/plain:
            schema:
              type: string
      ’400’:
        $ref: ’#/components/responses/BadRequest’
      ’401’:
        $ref: ’#/components/responses/Unauthorized’
      ’404’:
        $ref: ’#/components/responses/NotFound’

/scans/{id}/issues:
  get:
    summary: Receive a listing of all issues found in a scan
    description: >-
      This operation returns a list of issues that were found in a scan.
      If the scan did not find any issues, an empty array of items will
      be returned.
    security:
      - ApiKeyAuth: []
        ApiUser: []
    tags:
      - auth
    parameters:
      - $ref: ’#/components/parameters/scanIdParam’
    responses:
      ’200’:
        description: Successful transfer of a list of issues
        content:
          application/json:
            schema:
              $ref: ’#/components/schemas/ListOfIssues’
      ’400’:
        $ref: ’#/components/responses/BadRequest’
      ’401’:
        $ref: ’#/components/responses/Unauthorized’
      ’404’:
        $ref: ’#/components/responses/NotFound’

/scans/{id}/issues/{issueid}:
  get:
    summary: Receive a single issue from a scan
    description: >-
      This operation returns an issue object, which contains all information
      to describe an issue found in the vulnerability scan. This contains:

      - The scanner and its rule that found the issue plus infos on how to use
          it standalone

      - The type of the vulnerability that was found

      - The slug to an explanation object which describes the vulnerability.

      - An array of affected files containing:
        - The file path
        - An array of relevant lines in the file, including a description

      For detailed informations on how the issues object works, refer to the
      SCRAP documentation, that is linked in the `/` endpoint.
    security:
      - ApiKeyAuth: []
        ApiUser: []
    tags:
      - auth
    parameters:
      - $ref: '#/components/parameters/scanIdParam'
      - in: path
        name: issueid
        required: true
        schema:
          type: integer
    responses:
      '200':
        description: Successful transfer of an issue object
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Issue'
      '400':
        $ref: '#/components/responses/BadRequest'
      '401':
        $ref: '#/components/responses/Unauthorized'
      '404':
        $ref: '#/components/responses/NotFound'

/explanations:
  get:
    summary: Get a list of available explanations
    tags:
      - public
    description: >-
      This operation returns all explanations on code vulnerabilities
      that are available in SCRAP. If you want to exclude stub explanations
      which only provide links to further resources, use the `stub` parameter.
    parameters:
      - $ref: ’#/components/parameters/offsetParam’
      - $ref: ’#/components/parameters/limitParam’
      - in: query
        name: stub
        required: false
        schema:
          type: boolean
          default: true
        description: Set this to false, if you don’t want to exclude stub explanations
    responses:
      ’200’:
        description: Successful transfer of a list of explanations
        content:
          application/json:
            schema:
              $ref: ’#/components/schemas/ListOfExplanations’
      ’400’:
        $ref: ’#/components/responses/BadRequest’

/explanations/{slug}:
  get:
    summary: Retrieve an explanation to a vulnerability
    description: >-
      This operation returns an explanation object, which contains a description
      of a vulnerability and information on how to fix it. Additional ressources
      may be linked to in the `references` array.
    tags:
      - public
    parameters:
      - in: path
        name: slug
        required: true
        schema:
          type: string
    responses:
      ’200’:
        description: Successful transfer of an explanation object
        content:
          application/json:
            schema:
              $ref: ’#/components/schemas/Explanation’
      ’400’:
        $ref: '#/components/responses/BadRequest'
      '404':
        $ref: '#/components/responses/NotFound'

/scanners:
  get:
    summary: Retrieve a list of available scanners
    description: >-
      This operation retrieves a list of all the scanners that are used by
      the SCRAP server to scan for vulnerabilities. The list consists
      of the full scanner objects. At the moment, with only a few scanners
      deployed, there is no need for an extra endpoint to retrieve single
      scanners.
    tags:
      - public
    responses:
      '200':
        description: Successful transfer of a list of scanners
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Scanner'


components:
parameters:
  offsetParam:
    in: query
    name: offset
    required: false
    description: The (0-indexed) number of the first item to retrieve. Only
        use in combination with limit.
    schema:
      type: integer
      minimum: 0
      default: 0
  limitParam:
    in: query
    name: limit
    required: false
    schema:
      type: integer
      minimum: 1
      default: 20
    description: The amount of items to retrieve starting from offset.
  scanIdParam:
    in: path
    name: id
    required: true
    schema:
      type: string
      format: UUID
    description: The UUID of a previously submitted scan.
  filePathParam:
    in: path
    name: filepath
    required: true
    schema:
      type: string
      format: uri
    description: >-
      The path of a file within a scan. Either the uploaded filename itself,
      or, if an archive was uploaded, the path of the file relative to the
      archive’s root directory.

responses:
  NotFound:
    description: The specified resource was not found
    content:
      application/json:
        schema:
          $ref: ’#/components/schemas/Error’
        example:
          error:
            code: "404"
            message: "The [item] you requested does not exist."
            additionalInfo: "Use /[item] to get a list of scans."
  Unauthorized:
    description: You are not authorized to access this resource
    content:
      application/json:
        schema:
          $ref: ’#/components/schemas/Error’
        example:
          error:
            code: 401
            message: "You are not authorized to access this resource"
  BadRequest:
    description: >-
      At least one of the request parameters was malformed or the request
      is otherwise not valid.
    content:
      application/json:
        schema:
          $ref: ’#/components/schemas/Error’
        example:
          error:
            code: 400
            message: "The parameter you provided is not valid."
  FileTooBig:
    description: >-
      The uploaded file was too big. Especially if there is a ‘public‘ API key
         in
      use, these limits might be rather low. Consult the servers documentation
      on if and how big the upload limits are.
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Error'
        example:
          error:
            code: 413
            message: "The uploaded file is too big."
            additionalInfo: "Public users only have file size limits for their
                scans."
  WrongFileType:
    description: >-
      SCRAP will only accept either single PHP files or
      gzipped tar archives, containing at least on .php file.
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Error'
        example:
          error:
            code: 415
            message: "The file type you provided is not valid."
            additionalInfo: "User either .pdf or .tgz files."

schemas:
  Meta:
    type: object
    properties:
      api:
        type: string
        description: Name of the API
      version:
        type: string
        description: Version of the API
      openapi_file:
        type: string
        format: path
        description: Path to a YAML representation of the API definition
      definition:
        type: string
        format: uri
        description: Link to an API desription, such as provided by Swagger UI
      documentation:
        type: string
        format: uri
        description: Link to the documentation of the SCRAP project
    example:
      api: "scrap"
      version: "1.0.0"
      openapi_file: "/static/scrap_api.yaml"
      definition: "https://app.swaggerhub.com/apis/tantemalkah/SCRAP/1.0.0"
      documentation: "https://scrap.tantemalkah.at/docs"

  Paging:
    type: object
    properties:
      count:
        type: integer
        minimum: 0
      next:
        type: string
        format: uri
      previous:
        type: string
        format: uri

  Error:
    type: object
    properties:
      error:
        type: object
        properties:
          code:
            type: string
          message:
            type: string
          additionalInfo:
            type: string
        required:
          - code
          - message

  ListOfScans:
    type: object
    properties:
      paging:
        allOf:
          - $ref: '#/components/schemas/Paging'
        example:
          count: 23
          next: /api/v1/scans?limit=5&offset=10
          previous: /api/v1/scans?limit=5&offset=0
      items:
```

```
        type: array
        items:
          type: string
          format: uuid
        example:
          - "4fb9e66e-67a8-11ea-a2eb-983b8fc20c86"
          - "16f324cc-2abb-455d-9561-7c460840b90a"
          - "006739a6-66cf-4790-a89d-1bc60634e2c9"
          - "2d37480c-67a8-11ea-a2eb-983b8fc20c86"
          - "462e3fbe-67a8-11ea-a2eb-983b8fc20c86"

  Scan:
    type: object
    properties:
      id:
        type: string
        format: uuid
      status:
        type: object
        properties:
          stage:
            type: string
            enum: [loading, pending, analysing, done]
            description: Describes the current stage in the analysis pipeline
          percentage:
            type: integer
            minimum: 0
            maximum: 100
            description: Describes the progress in the current stage. Always
               100 for _done_, and 0 for _pending_
      issuesFound:
        type: integer
        minimum: 0
        description: Number of issues that where found in this scan.
      files:
        type: integer
        minimum: 1
        description: >-
          Number of files contained in the uploaded package.
          1, if only a single file or a .tgz containing a single file was
              uploaded.
      created:
        type: string
        format: date-time
        description: Time when the file/package to scan was uploaded
      analysed:
        type: string
        format: date-time
        description: Time when the scan analysis was completed

  Scanner:
    type: object
    properties:
      name:
        type: string
        description: "The name of the scanner"
      slug:
        type: string
        description: "The slug representation of the scanner, which is used in
            issue objects"
      version:
        type: string
        description: "The version number of the scanner, that is deployed in
            the SCRAP server"
      uri:
        type: string
        format: uri
        description: "A link to the scanners web site or repository"
      comment:
        type: string
        format: markdown
        description: "An optional comment describing how the scanner is used
            in SCRAP"
    example:
      name: "PHP_CodeSniffer"
      slug: "phpcs"
      version: "3.5.4"
      uri: "https://github.com/squizlabs/PHP_CodeSniffer"
      comment: "Using the [phpcs-security-audit
          v2](https://github.com/FloeDesignTechnologies/phpcs-security-audit)"

  ListOfIssues:
    type: object
    properties:
      paging:
        allOf:
          - $ref: '#/components/schemas/Paging'
        example:
          count: 23
          next: /api/v1/scans/42/issues?limit=3&offset=6
          previous: /api/v1/scans/42/issues?limit=3&offset=0
      items:
        type: array
        items:
          type: object
          properties:
            id:
              type: integer
              minimum: 0
            type:
```

```yaml
                type: string
          example:
            - id: 0
              type: "SQLi"
            - id: 1
              type: "XSS"
            - id: 2
              type: "Remote File Inclusion"

    Issue:
      type: object
      properties:
        source:
          type: object
          properties:
            scanner:
              type: string
              description: The slug of the scanner from which this issue was generated
            rule:
              type: string
              description: The specific rule of the scanner, that triggered this issue
            info:
              type: string
              format: uri
              description: Link to a page with additional information on how to use the scanner on its own
            cli:
              type: string
              description: Command line that the SCRAP server used to analyse the file(s)
          required:
            - scanner
            - rule
          example:
            scanner: "phpcs"
            rule: "Security.BadFunctions.Mysqli.WarnMysqlimysqli_query"
            info: "https://github.com/FloeDesignTechnologies/phpcs-security-audit"
            cli: "php scanners/phpcs/phpcs.phar --standard=scanners/phpcs-sa/Security -s --report=json scrap-upload-tmp"
        type:
          type: string
          description: What type of vulnerability was found
          example: "SQLi"
        explanation:
          type: string
          description: Slug to an explanation.
          example: "sqli_mysqli_dynamic_param"
        affectedFiles:
          type: array
          items:
            type: object
            properties:
              path:
                type: string
                format: uri
              lines:
                type: array
                items:
                  type: object
                  properties:
                    num:
                      type: integer
                      minimum: 0
                    linkedTo:
                      type: integer
                      minimum: 0
                    characters:
                      type: object
                      properties:
                        from:
                          type: integer
                          minimum: 0
                        to:
                          type: integer
                          minimum: 0
                    text:
                      type: string
                    description:
                      type: string
                  required:
                    - num
                    - characters
          example:
            - path: "index.php"
              lines:
                - num: 23
                  characters:
                    from: 5
                    to: 63
                  text: "$r = mysqli_query($conn, 'SELECT * FROM posts WHERE id
                      = '.$id);"
                  description: "MYSQLi function mysqli_query() detected with
                      dynamic parameter"

    ListOfFiles:
      type: object
      properties:
        paging:
          $ref: '#/components/schemas/Paging'
        items:
          type: array
          items:
            type: string
            format: uri
            description: Link to the file's blob endpoint
      example:
        paging:
          count: 3
          next: ""
          previous: ""
        items:
          - "/api/v1/scans/4fb9e66e-67a8-11ea-a2eb-983b8fc20c86/files/index.php"
          - "/api/v1/scans/4fb9e66e-67a8-11ea-a2eb-983b8fc20c86/files/style.css"
          -
             "/api/v1/scans/4fb9e66e-67a8-11ea-a2eb-983b8fc20c86/files/database.php"

    File:
      type: object
      properties:
        path:
          type: string
          format: uri
          description: >-
            File path in relation to the submitted scan root.
            If a single file was submitted, then the path is the filename.
        contentType:
          type: string
          format: media-type
          description: >-
            Media type of the file, as
            [defined by
                IANA](https://www.iana.org/assignments/media-types/media-types.xhtml).
            PHP files will receive a media type of `application/x-php`, as there
            is no standardised media type for PHP files yet.
        size:
          type: integer
          minimum: 0
          description: Size of the file in byte.
        blob:
          type: string
          format: uri
          description: Link to the `blob` endpoint to download the file
      example:
        path: "index.php"
        type: "application/x-php"
        size: "3096"
        blob: "/api/v1/scans/4fb9e66e-67a8-11ea-a2eb-983b8fc20c86/blob/index.php"

    ListOfExplanations:
      type: object
      properties:
        paging:
          allOf:
            - $ref: '#/components/schemas/Paging'
          example:
            count: 23
            next: /api/v1/explanations?limit=1&offset=6
            previous: /api/v1/explanations?limit=1&offset=4
        items:
          type: array
          items:
              $ref: '#/components/schemas/ShortExplanation'

    ShortExplanation:
      type: object
      properties:
        name:
          type: string
        slug:
          type: string
        type:
          type: string
      example:
        name: SQL Injection through unsanitized `id` parameter
        slug: sqli_unsanitized_id
        type: SQLi

    Explanation:
      type: object
      properties:
        name:
          type: string
        slug:
          type: string
        type:
          type: string
        isStub:
          type: boolean
        shortDescription:
          type: string
          format: markdown
        longDescription:
          type: string
          format: markdown
        howToFix:
          type: string
```

~~~
          format: markdown
        references:
          type: array
          items:
            type: string
            format: uri
      example:
        name: SQL Injection through unsanitized `id` parameter
        slug: sqli_unsanitized_id
        type: SQLi
        isStub: true
        shortDescription: >
          If you use an `id` parameter without validation in an unparameterised
              SQL
          query, an attacker can easily inject malicous code.
        longDescription: |
          If you use an `id` parameter without validation in an unparameterised
              SQL
          query, an attacker can easily inject malicous code.

          __What does this mean?__

          If you take for example the following PHP code:
          ```php
          $id = $_GET["id"];
          # do some other stuff
          # and then query for, e.g. a user with this id in the database:
          $query  = "SELECT first_name, last_name FROM users WHERE user_name =
              '$id';";
          $result = mysqli_query($connection, $query);
          ~~~
          What would happen, if someone submits `1' OR 1=1; -- -` as a value?
          This would lead to the following effective query:
          ```sql
          SELECT first_name, last_name FROM users
            WHERE user_name = '1' OR 1=!; -- -'
          ```
          As the `-- -` makes the reminder of the original query (in this case
              only)
          the `'`, we have a new query, with a `WHERE` clause that is always
              true.
          Therefore not only one row for a specific user will be returned, but
              all
          users.
          But worse could be done, e.g. by using the `UNION` construct to find
              out
          about other tables data or even the whole database scheme.
        howToFix: |
          One of the best ways in PHP to safeguard against SQL injections is to
```
```

~~~
        use [prepared
            statements](https://www.php.net/manual/en/mysqli.quickstart.prepared
            -statements.php). Instead of putting the parameters into the
        query yourself, you can let the database library do that for you with
        the `prepare` method of a mysqli database object:
        ```php
        $db = new mysqli("example.com", "user", "password", "database");

        # do some other stuff

        # STEP 1: prepare the query statement
        $stmt = $db->prepare('SELECT first_name, last_name ' .
                             'FROM users WHERE user_name = ?');
        if (!$stmt) {
          echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
        }

        # STEP 2: bind the parameter to the query statement
        if (!$stmt->bind_param("i", $id)) {
          echo "Binding parameters failed: (" . $stmt->errno . ") " .
              $stmt->error;
        }

        # STEP 3: execute the query
        if (!$stmt->execute()) {
          echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
        }
        ```
        Apart from using such prepared statements it is also always advisable
        to [validate your user
            inputs](https://www.w3schools.com/php/php_form_validation.asp). You
            can also use the [PHP filter
            functions](https://www.w3schools.com/php/php_ref_filter.asp)
        to check if the input conforms to what you expect.
      references:
        - https://www.w3schools.com/sql/sql_injection.asp
        - https://www.php.net/manual/en/security.database.sql-injection.php
        - https://en.wikipedia.org/wiki/SQL_injection
        - http://cis1.towson.edu/~cssecinj/modules/other-modules/database/sql-
            injection-introduction/
        - https://xkcd.com/327/
        - https://bobby-tables.com/
        - https://owasp.org/www-community/attacks/SQL_Injection

securitySchemes:
  ApiKeyAuth:
    type: apiKey
    in: header
    name: X-API-KEY
  ApiUser:
~~~

```
      type: apiKey
      in: header
      name: X-API-USER
```

Code 40: Listings/scrap_api.yaml