SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2019; **00**:1–39
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Sensei: Enforcing Secure Coding Guidelines in the IDE

Pieter De Cremer[*], Nathan Desmet[†], Matias Madou[‡], Bjorn De Sutter[§]

SUMMARY

We discuss the potential benefits, requirements, and implementation challenges of a security-by-design approach in which an integrated development environment (IDE) plugin assists software developers to write code that complies with secure coding guidelines. We discuss how such a plugin can enable a company's policy-setting security experts and developers to pass their knowledge on to each other more efficiently, and to let developers more effectively put that knowledge into practice. This is achieved by letting the team members develop customized rule sets that formalize coding guidelines and by letting the plugin check the compliance of code being written to those rule sets in real time, similar to an as-you-type spell checker. Upon detected violations, the plugin suggests options to quickly fix them and offers additional information for the developer. We share our experience with proof-of-concept designs and implementations rolled out in multiple companies, and present some future research and development directions. Copyright © 2019 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: security by design; software development; IDE support; coding guidelines

1. INTRODUCTION

Security is a major concern in software development: 90 percent of security issues are caused by problems or oversights in underlying code [1]. They are the results of mistakes made by the programmers during development. Sometimes this is out of lack of knowledge or awareness of the security implications of their work. Other times the application developers have sufficient knowledge but are not able to apply it in practice for various reasons [2]. In many software projects, application programming interface (API) knowledge and security knowledge are not uniformly distributed among the developers. A developer might have encountered a security problem before, researched its solution and maybe even documented it. But in practice this knowledge is not sufficiently distributed and other developers spend additional time to resolve similar issues. When developers create new APIs they usually include documentation on its proper usage. But these guidelines are often ignored and rarely enforced. At the same time, application security experts are understaffed with an average of one security expert per 75 developers [3].

In an attempt to scale security solutions better, companies use tools to automate security testing in their development process. Penetration testing and code analysis are common practices in many

[*]pdecremer@securecodewarrior.com
[†]ndesmet@securecodewarrior.com
[‡]mmadou@securecodewarrior.com
[§]bjorn.desutter@ugent.be
[*]Correspondence to: Secure Code Warrior, Baron Ruzettelaan 5/3, 8310 Assebroek, Belgium
or Computer Systems Lab, Ghent University, Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium

Contract/grant sponsor: Flanders Innovation and Entrepreneurship Agency (VLAIO)

companies to detect security problems, including both (implementation) bugs and (design) flaws. Bugs are implementation problems caused by mistakes by the developer, flaws are issues that are manifested in implementation but stem from design. These practices tackle the problem with a testing approach, trying to find problems after they have been introduced, rather than by preventing them. Experience has shown that security should not be an afterthought of software development. It should be addressed earlier in the development to minimize cost and time invested [4, 5, 6, 7], and to improve security. In line with the security-by-design mantra, a movement is ongoing towards techniques that try to identify possible security problems as early as possible in the software development life cycle (SDLC). Still, those approaches all tackle SDLC phases after the initial development. They hence involve all kinds of costs that should be considered but are hard to measure, such as the time it takes for developers to make the context switch from their primary task of developing new functionality to their additional task of fixing security issues identified in code that they or others wrote hours, days, or longer before.

In this paper we introduce a fundamentally different approach: Instead of identifying problems as early as possible after the initial development, such as during the build process, we propose to identify potentially problematic code fragments as they are written. Our approach is based on enforcing secure coding guidelines that, when adhered to, prevent the introduction of vulnerabilities. These coding guidelines are API-level limitations and instructions. This means that the focus of this approach is on the prevention of security bugs introduced during implementation, rather than security flaws that stem from mistakes in the design. To implement this approach, the developer's integrated development environment (IDE) monitors the use of APIs that are the subject of such guidelines, flags violations thereof, and proposes fixes.

This approach allows any team member to create a new coding guideline to be enforced in the software project. This empowers security experts in the team to distribute their knowledge in the form of guidelines and to monitor how well non-experts take up the knowledge. It also facilitates the distribution of knowledge gathered by other team members.

In support of this approach, we have developed a plugin for various IDEs. It is designed to help companies adapt the approach of customizing and enforcing coding guidelines. The plugin can be compared to a conventional spell checker in a word processor: It checks the input of the user against a set of rules, flags violations, and suggests possible fixes. Various iterations have been rolled out to companies. This allows us to pinpoint the critical aspects for successful roll-outs.

In our current implementation we are able to support memory safe languages. Frameworks developed in memory safe languages, such as Java, in our experience have more manifest APIs than those in unsafe languages like C. As a result the intent of the developer is more easily detected with local analyses in Java. Hence our initial focus on it.

The contributions of this work start with an argumentation in Section 2 for the requirements and goals that need to be considered to make a successful IDE plugin to enforce coding guidelines. Section 3 discusses a proof of concept (PoC) of this plugin that was developed for several IDEs to support the aforementioned languages. In Section 4 we discuss in more detail some of the consequences of our design. We analyze what type of vulnerabilities are more challenging to cover and we address some important aspects to create effective guidelines and to improve the usability of the plugin. Furthermore an overview is presented of the challenges encountered during development and roll-out of the plugins into an industrial environment, and some potential solutions are discussed. Next, in Section 5 we briefly explain how this developer tool can also be used by management. Section 6 describes our experience with developing and rolling out the tool, and analyzes the factors that contribute or hamper its take-up and use. We highlight goals that are achieved and discuss where there is still room for improvement. Section 7 discusses additional related work, and compares other tools to the developed PoC. Finally, Section 8 draws conclusions and discusses some future work and interesting research directions.

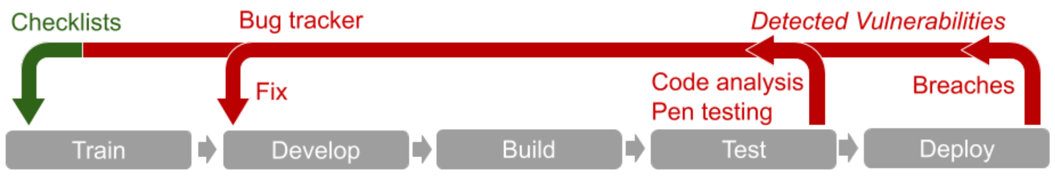


Figure 1. Testing approach to security.

2. GOALS AND REQUIREMENTS

Security issues still exist in most software products: 100% of the applications tested by Trustwave in 2017 displayed at least one vulnerability [8]. Many of these vulnerabilities are introduced by programmers during the development phase. Security and API knowledge is often distributed (unevenly) among members in the team, and security experts are understaffed. This means that many of the mistakes made by the developer are caused by a lack of knowledge or awareness.

Other times the mistakes are well known and the developer is sufficiently trained to understand them, but fails to apply this knowledge in practice. There are a number of factors contributing to this gap between a developer's knowledge and practice, as researched by Xie et al. [9].

The main goal of our approach is to make developers put their own knowledge and that of other team members into practice during the development phase without requiring unscalable, costly, iterative forms of collaboration and communication cycles between individual experts and developers. To achieve this, we put forward three concrete goals.

Firstly, the tools in support of the approach should be usable as early as possible in the SDLC, i.e., during the actual code development. Secondly, the tools should distribute the security knowledge of one developer or security expert to the rest of the team. Thirdly, the tools should close developers' gap between knowledge and practice, and help them to apply their knowledge, while minimizing the impact on the main daily tasks of a software engineer, i.e., the development of new functionality.

2.1. Early in the SDLC

For a long time, security has been considered a part of software testing [10]. Security was addressed in a reactive manner, from the end (right) of the SDLC, as shown in Figure 1. Based on vulnerabilities reported when the application was tested after its initial development was completed or when it was already deployed and in production, developer training was adapted, new coding checks were introduced, and bugs were fixed by revisiting code.

Experience has shown, however, that security should not be an afterthought of software development but that it should be addressed earlier in the development. This is not only to minimize costs [4, 5, 6, 11]. Shorter feedback loops also result in better learning performance [12, 13]. As a result a shift left movement is ongoing to try to identify possible security problems as early as possible in the SDLC, as illustrated in Figure 2. New project management techniques such as Agile and DevOps encourage fast incremental releases where the developer is also responsible for meeting non-functional requirements such as security. To support that, testing and deployment of security guidelines need to be more automated in short feedback loops, thus shifting security left.

While supporting the shift left, conventional vulnerability scanning tools still use a reactive, testing-based approach. Furthermore, in training developers are typically taught how certain mistakes lead to vulnerabilities, and how these can be exploited. Afterwards they are taught how to prevent the presented vulnerabilities. These practices are extended into the development phase, where the focus is still on the (sometimes complex) question of whether or not the code is vulnerable, and only if it is considered to be vulnerable, it becomes a candidate to be fixed. The shift left movement is certainly an improvement, but it is not yet perfect. Many security problems still occur. Companies acknowledge this, as is obvious from the incentives they put in place to minimize the impact of potential breaches, such as bug bounty programs.

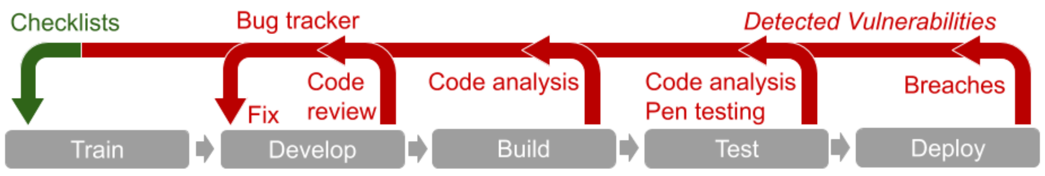


Figure 2. Shift left in the approach to security.

Many of the vulnerability scanning tools use complex control flow and data flow analyses to scan for vulnerabilities in the product. They identify, e.g., user input that is not properly validated and passed on to security-critical parts of the application. If it is determined that a malicious input exists that can cause unwanted or unexpected results, these issues are placed into the bug tracking system for developers to deal with. In order to successfully detect vulnerabilities, the calling context of routines needs to be known to perform the necessary global analyses. Because of this, such tools can only be deployed at a later stage in development. It is, in other words, not possible to shift even more to the left with only these tools. During the earlier development stages of a product, it is entirely possible that no user input can reach a buggy routine yet. The classic approach will only flag the routine once the context exists where it can be exploited. This then requires the developers to go back to secure the routine at a later time than when they were originally developing it.

In short, even in the ongoing shift left movement, the problem is still approached from the right side of the SDLC, following the detection of vulnerabilities. The detection is shifted as much to the left as possible but the approach is still reactive, and requires revisiting code (possibly long) after it has been developed.

In this paper, we propose to shift to the left even further, and actually start from the left. Concretely, we propose enforcing coding guidelines in the IDE, during the initial writing of code. Figure 3 illustrates how this approaches the problem from the left. The focus is on proactive secure software development; if and how vulnerabilities can be exploited becomes secondary to a developer during development. The goal is to enforce coding guidelines regardless of the context from which the code fragment being written will be invoked.

This approach can start in the training, by teaching the coding guidelines by means of small, context-free examples. Optionally, this training can be extended by explaining interested developers the raison d' être of the guidelines, which can be made clear by considering the examples in relevant context. Importantly, the broadening of the training scope is optional. It is not necessary to enable a developer to implement the guidelines. It is therefore easy to extend their acquired knowledge into practice, where the developer has to apply the learned lessons with minimal added cognitive effort. The benefits of adhering to coding guidelines earlier in the SDLC to minimize risks have long been documented [14]. Our contribution is a method and tools to enforce that adherence as earlier as possible in the SDLC.

Because our goal is to enforce coding guidelines regardless of the context in which a code fragment being written will ever be invoked, both the formulation of the guidelines and their enforcement need to be based on local scopes only. The coding guidelines that we envision to be enforced are API-level guidelines such as misused method calls, missing preceding or succeeding method calls, missing or faulty method parameters, or incomplete configuration files. These types of guidelines meet our requirement of needing only local, short-running analyses to check their deployment; their calling context is indeed irrelevant. Feedback loops can then become even shorter. Similar to an as-you-type spell checker, developers making security mistakes will be aided in real time, i.e., as soon as they write the problematic code. Using this approach we help developers writing secure code from the start, thus truly starting security from the left.

Another advantage of ignoring context and enforcing coding guidelines at all times is protection for future use. We already stated that the classic approach only flags a routine once the context exists in which it can be exploited and that this requires the developers to go back to secure routines at a

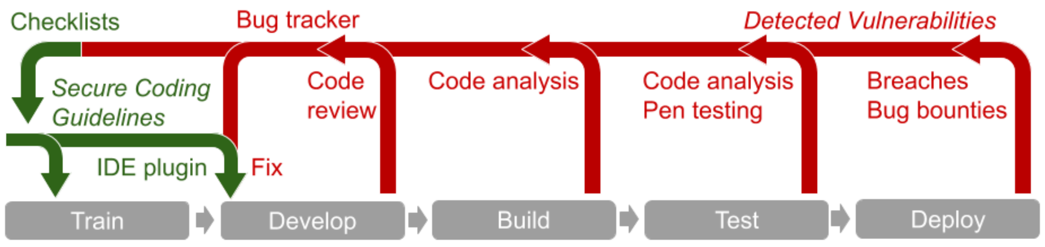


Figure 3. Security approached from the left.

later time than when they were originally developing them. It is also possible that at a certain point in time all user input that can reach a routine gets validated correctly. The classic approach then does not flag this routine as it cannot be exploited. However, at a later stage the routine could be reused in a different context where unvalidated user input is used and the vulnerability can eventually be exploited. When coding guidelines are enforced by an IDE, problematic routines are flagged during their development. Since the context is not yet known, or ignored, sometimes coding guidelines are enforced where their violation is not (yet) leading to a vulnerability. This is intentional in our approach, as the goal is precisely to secure the routine by design, including for future uses.

In many software companies code bases are too large for global analyses that require the entire calling context to be available. Tools can often not access the entire source code, as this is often too much data for a single machine [15]. This is another reason why conventional static analysis solutions are not performed in the development phase but are often integrated in the build or test phase by use of continuous integration and continuous delivery (CI/CD) tools. As a result of using local analyses to enforce coding guidelines early in the SDLC, our approach makes it possible to perform analyses on the developer's machine. It also becomes feasible to perform them with a latency lower than 125 ms. This means that the code produced by a developer is analyzed in real time, with the same latency as an as-you-type spell checker in a word processor, even on large code bases.

A final advantage of using the developer's IDE to enforce secure coding guidelines is that it minimizes the developer's distraction from his main tasks, i.e., developing new functionality. The developer already uses an IDE during that stage of development, and is already used to seeing certain syntactical and functional errors being flagged in his IDE, and knows how to handle these in a timely manner. This provides a perfect fit, and provides a natural basis to extend the functionality of the IDE to detect coding guideline violations as well.

2.2. *Knowledge Distribution*

Coding guidelines are a natural result of knowledge sharing. Consider the case where some security-minded developers research a security vulnerability. They might have spotted it themselves or have been made aware through the report of an analysis or penetration test. To resolve the issue, they dig through the API documentation or even test configurations manually until they can patch the vulnerability. To help other developers in their team, that might have expertise in security, identify and avoid similar mistakes, it is typically much easier to explain how to configure or how to use the API in order to prevent the vulnerability, rather than to explain how the prevented attack would have worked. In other words, sharing a coding guideline with non-experts is more productive than sharing a description of a type of attack to prevent.

Consider for example the case of Structured Query Language (SQL) injection. The developers research different ways of creating SQL queries and finds in the API documentation that input is automatically escaped when using parameterized queries. To share this knowledge they do not tell their peers "Do not allow SQL Injection". Instead they tell them "Write SQL queries using parameterized queries". An important requirement of the tool is hence the possibility to easily create new coding guidelines to be enforced.

2.3. Applying Knowledge in Practice

Few existing tools provide mitigation techniques to resolve identified issues, and those that do generally serve generic information to educate the developers on potential attacks, but not quick fixes relevant and directly applicable to the code at hand. Most tools provide no mitigation at all [16]. Instead, when issues are detected by the tools or tests, they are filed in the bug tracker, after which it is up to the developers to dive back into the code and secure the routine. Not only do the developers need to reverse-engineer part of the context and code, they also need to understand the sometimes complex issues regarding security. Some tools provide extensive info for doing so, others do not [16]. Ultimately it is still up to the developer to resolve the issue, which can result in sub-optimal solutions. For example, when a developer performs custom escaping of dangerous characters before executing a SQL query, this can completely prevent SQL injection. However, this is not the desired solution from a security perspective. Instead, it is usually advised to use parameterized queries to prevent SQL injection[‖].

Even when developers are knowledgeable enough and they receive enough information from the tools, they often do not address non-functional errors unless the effort and cognitive burden is sufficiently small [17]. Moreover, development does not stop, new issues are introduced daily and pile up in the bug trackers. Essentially, tools have been created to detect issues, but not enough support exists to actually fix them. We conclude that another important requirement for the tool is to provide easily applicable quick fixes to the developer, both to keep the cognitive burden of fixing issues small, and to ensure that issues are fixed properly and timely.

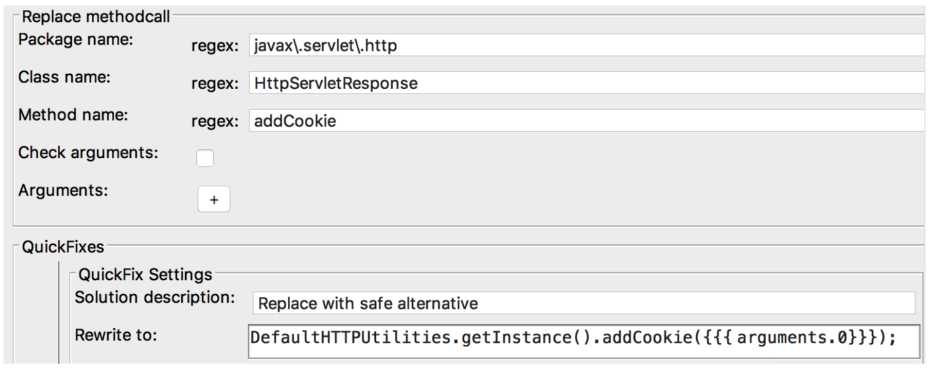
Some code review tools such as Tricorder [15] and Snyk, which are both discussed in more detail in Section 7, do provide quick fixes during review. However, review tools do not always provide syntax highlighting and checking. More importantly code transformations are complex. Occasionally, transformations presented as quick fixes do not fully apply or require manual completion. Sometimes multiple quick fix solutions are applicable to resolve an issue, in which case a developer needs to make an actual decision and express it in some form. In all of those cases, the developer still needs to revisit the code with another tool, which requires time and energy consuming context switches. As evident from previous research, many authors choose to fix the code in their own editor rather than use a code review tool for this purpose [15, 18]. A more fluent interaction with developers is therefore to be preferred instead.

This highlights the importance of applying quick fixes in the IDE when the developer is still working on the code at hand. The developer can then more easily complement and extend incomplete quick fix transformations and make decisions on the preventive measure without much additional cognitive effort or switching contexts. When coding guidelines are enforced, it is also more clear to a developer what should be done to comply, since any ill-favored, ad-hoc solutions will still be flagged as violations of the coding guideline. As a result, the preferred solutions are more likely applied uniformly across a project.

3. DESIGN AND IMPLEMENTATION

In line with the described goals and requirements, we developed a tool since Nov 2015. The result is an IDE plugin called Sensei. We started to collect customer data since our first trial with a customer with 800 developers in March 2016. The first versions of the plugin were distributed to a few select customers to thoroughly test them before a full release. These customers needed to be application security savvy, such that we get accurate feedback on what works and what does not work. The try-outs with customers were initially limited to 10 people per try-out group to allow for short feedback

[‖]https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_1:_Prepared_Statements_.28with_Parameterized_Queries.29



Replace methodcall

Package name:	regex:	javax\.servlet\.http
Class name:	regex:	HttpServletResponse
Method name:	regex:	addCookie
Check arguments:	☐	
Arguments:	+	

QuickFixes

QuickFix Settings

Solution description:	Replace with safe alternative
Rewrite to:	`DefaultHTTPUtilities.getInstance().addCookie({{{ arguments.0}}});`

Figure 4. Input fields of old the model-based rule editor.

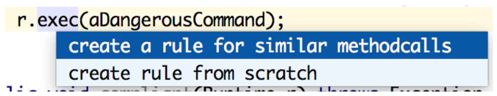
and implementation cycles. The current stage of rolling out is within groups of 100+ developers. The next phase is rolling it out company-wide where some of our customers have 10.000+ developers. A tool like ours requires an IDE that builds an abstract representation of the code and facilitates analyses on it. Popular IDEs like JetBrains IntelliJ IDEA, Android Studio, and Eclipse for Java and Visual Studio for C# offer the necessary functionality, but light-weight IDEs such as vim do not. For Java, the first supported language, coding guidelines have been created with framework support for Java EE, Java Spring and Java Android, in order to test the effectiveness. As a result a total of over 300 API-level rules can be used, in addition to customers' own rules. Feedback following the uptake of various generations of the tool provided useful insights. In this section, we discuss design, implementation, and usability decisions. We document the most important choices we had to make, lessons we learned along the way, evolving designs, and the most important aspects to improve usability in the field.

3.1. *Creating Rules*

One of the requirements of the tool is to share knowledge easily. To meet that requirement customization and distribution of the rules is crucial. The rules customization should be scalable. It should hence not be a service provided by us. It should allow security experts in the companies to distribute their guidelines related to their security-critical concepts, but also allows developers to share more project-specific or team-specific guidelines among each other. For this reason the rule creation should be easy and fast, and at the same time versatile.

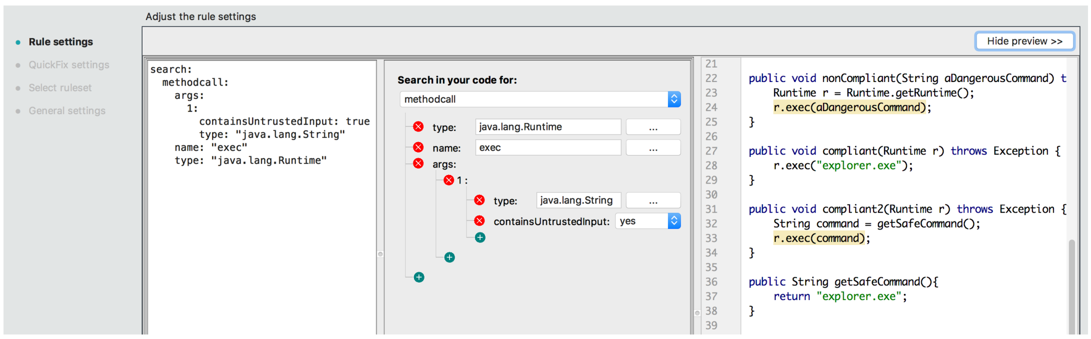
Our first approach to let users create new rules used predefined rule models, and let the rule writer fill in a number of fields. A simple example of such a model is "Replace method call model". Figure 4 shows a rule being created to replace the `addCookie` method with a safe alternative from the Open Web Application Security Project (OWASP) Enterprise Security API (ESAPI), an open source, web application security control library designed to make secure development easier [19]; the organization also provides some commonly used security guidelines. The rule writer has to fill in some specifics about the method that should be marked, such as the package, class, and method names. Then they can write one or more quick fixes. Here they have to create a quick fix description and they have to define the code fragment that needs to be put in place. To do so they can reuse arguments, method names, and more from the original code by means of a template language. In the field "Rewrite to" the example quick fix reuses the first argument in the original method call.

However, for more complex models the number of input fields grew rapidly to accommodate a plethora of corner cases, and so did the number of models for multiple scenarios. As of now the old rule editor has over 40 different models. With this many models, it becomes overwhelming for rule writers that have to select a model to enforce their desired coding guideline. The described model-based rule creation process is not flexible and intuitive enough, so in the next iteration a new approach was taken.



```
r.exec(aDangerousCommand);
```

Figure 5. Context-aware rule creation menu.



```
search:
  methodcall:
    args:
      1:
        containsUntrustedInput: true
        type: "java.lang.String"
    name: "exec"
    type: "java.lang.Runtime"
```

Search in your code for: methodcall

- type: java.lang.Runtime
- name: exec
- args:
  - 1:
    - type: java.lang.String
    - containsUntrustedInput: yes

```
public void nonCompliant(String aDangerousCommand) t
    Runtime r = Runtime.getRuntime();
    r.exec(aDangerousCommand);
}

public void compliant(Runtime r) throws Exception {
    r.exec("explorer.exe");
}

public void compliant2(Runtime r) throws Exception {
    String command = getSafeCommand();
    r.exec(command);
}

public String getSafeCommand(){
    return "explorer.exe";
}
```

Figure 6. The rule creation wizard with a rule automatically generated from the context.

In the new approach we split up the rule in two parts: A trigger to identify the violation, plus an optional quick fix to correct the vulnerability consistently according to company best practices. Triggers are now specified by way of YAML Ain't Markup Language (YAML) syntax (https://yaml.org/), which provides more flexibility. Since this requires the rule writers to learn this syntax, we have provided some tools to assist them, in the form of a Graphical User Interface (GUI) that can be used to build the desired rule from scratch. In addition, the rule editor is now more context-aware. The rule writer can open a rule creation wizard by pressing a key combination in the text editor in the IDE and selecting "create new rule". This opens a context-aware menu depending on the position of the cursor. For example, if the cursor was on a method call, the menu contains an option to create a new rule for similar method calls, as shown in Figure 5.

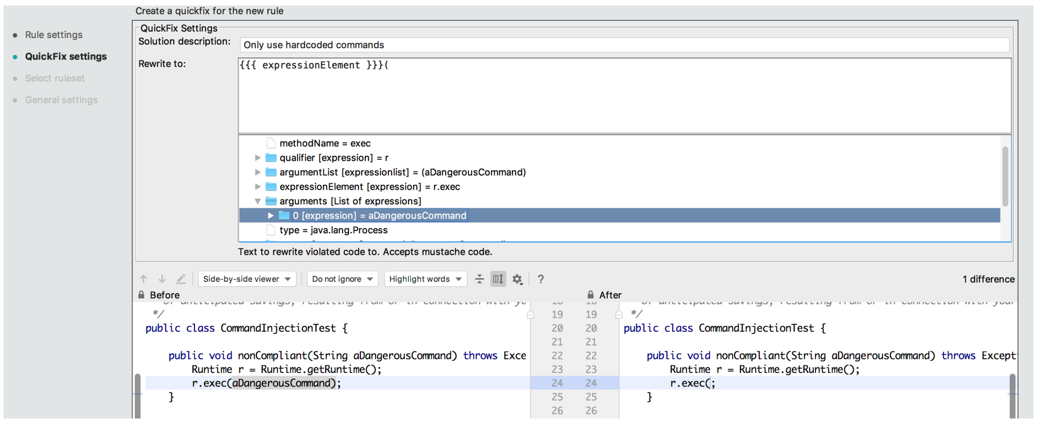
When this context-aware option is chosen, the rule creation wizard is opened and a rule is automatically suggested from the available context. Some examples of context that can be pre-filled are the type and the name of method calls, arguments, and instance creations. The user can then adjust the rule to reach the desired results through the YAML code or the provided GUI. This window (Figure 6) also provides a preview panel. In this panel the code file from which the rule wizard was opened is shown, as well as the effects of the rule, which allows for easy customization.

After creating a trigger, the rule wizard creates an optional quick fix. Here the developer has to fill in the solution description and the replacement code. For the replacement code they can make use of the same template language as in the first approach to reuse parts of the original code. Below the input field an overview is provided of the available parts of the original code, as shown in Figure 7, and a live preview is shown (in the lower right corner).

We have observed that the context-aware rule wizard has greatly sped up the rule creation process. In practice, creating new rules often starts from a bad code example, either when fixing a vulnerability or while reviewing the code of a colleague. The rule writer can then simply start the rule creation wizard from this example. The live previews also greatly improve the usability, since before they were introduced, to create a finished rule the rule writer was required to go back and forth several times to test the rule in the IDE and adjust it in the rule editor.

3.2. *Verifying Rules*

To check the rules, our tool reuses the IDE syntax checking features. When a developer writes new code, the IDE rebuilds the Abstract Syntax Tree (AST) and computes the changes compared to the previous version. A limited AST of the changes, containing the necessary symbol information, is then passed on, allowing tools to only analyze the changes. On this AST a combination of



Create a quickfix for the new rule

QuickFix Settings

Solution description: Only use hardcoded commands

Rewrite to:

```
{{{ expressionElement }}}(
```

```
methodName = exec
▶ qualifier [expression] = r
▶ argumentList [expressionlist] = (aDangerousCommand)
▶ expressionElement [expression] = r.exec
▼ arguments [List of expressions]
    ▶ 0 [expression] = aDangerousCommand
type = java.lang.Process
```

Text to rewrite violated code to. Accepts mustache code.

```
*/
public class CommandInjectionTest {

    public void nonCompliant(String aDangerousCommand) throws Exce
        Runtime r = Runtime.getRuntime();
        r.exec(aDangerousCommand);
    }
```

```
*/
public class CommandInjectionTest {

    public void nonCompliant(String aDangerousCommand) throws Except
        Runtime r = Runtime.getRuntime();
        r.exec(;
    }
```

Figure 7. The fix creation wizard, with quick fix work in progress.

specialized light-weight versions of existing analysis techniques are used such as taint analysis, data flow analysis, and control flow analysis to verify the rules in real time.

3.3. *Managing Rules*

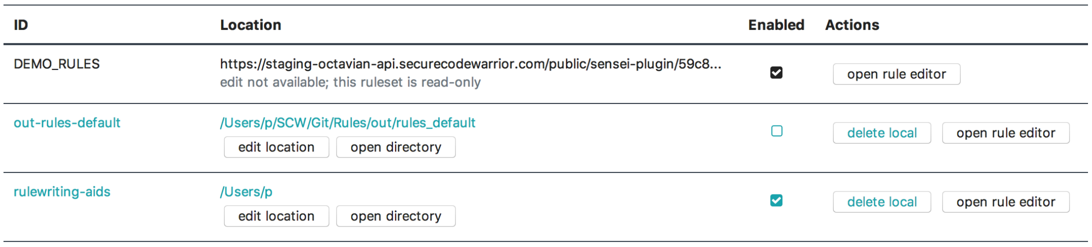
The rules are slowly gathered over time. When developers address security problems that are discovered by security testing or reported through bug bounty problems, it can be part of their task to create a rule that prevents this same vulnerability from occurring in the future. Security experts currently guide the developers by providing them with informal guidelines and checklists. These instructions sometimes use security jargon that might not be clear to all developers, and even if they are understood, that does not guarantee the developer will be able to apply them in practice. To help improve this communication, security experts can develop Sensei rules as well.

Generally we recommend to companies to start with no rules and use existing data on the security of their project as a starting point. This could be the report of a penetration test, or results of vulnerability scans. While resolving these issues in the code, developers and security experts can start building the first rules. Some clients are hesitant to start with an empty security tool and, despite our recommendations to customize rules for each project, still wish to receive a starting rule pack. For this reason we have created a small set of unopinionated, generally applicable rules, which we call the basic protection set. These rules aid in correctly using the standard libraries (e.g., Java EE). This set can be used as inspiration and to get both security experts and developers accustomed to the tool, but usually it does not flag many issues. Currently we have not yet developed more elaborate rule sets to cover common frameworks or libraries, because this would require too much upkeep for us. In the future we would encourage open-source rule sets for that purpose, to be developed and maintained together with the libraries.

Considering the different sources of rules, developers at our clients often have rules imposed by management and/or by the security team, as well as rules distributed among the developers per team or project. On top, there is the basic protection set provided by us as a starting point during the pilot program. To make the management of rules easier, we group rules into rule sets. Instead of distributing rules one-by-one, this allows for grouping and distributing related rules more easily.

In order to manage these rule sets in the IDE, a rule set manager window is provided, as shown in Figure 8. Each rule set is specified by an id and a location. The developer can enable or disable any rule set as well as edit rules in some rule sets.

Rule sets can be stored locally or remotely. Remote rule sets (shown in black) can be loaded from a github project (e.g., `git@gitserver:rulesrepos|master|myrules`) or a server (e.g., `https://remoteserver.com/myrules.zip`). Remote rule sets are only recommended to distribute imposed rule sets, since remote rules are not editable by the developers and are instead read-only. This allows management or the security team to manage the rules. Any updates to the remote rules are automatically pushed to all the developers. Locally stored rule sets (in blue font) are editable



ID	Location	Enabled	Actions
DEMO_RULES	https://staging-octavian-api.securecodewarrior.com/public/sensei-plugin/59c8... edit not available; this ruleset is read-only	✓	open rule editor
out-rules-default	/Users/p/SCW/Git/Rules/out/rules_default edit location open directory	☐	delete local open rule editor
rulewriting-aids	/Users/p edit location open directory	✓	delete local open rule editor

Figure 8. The rule set manager containing one remote and two local rule sets.

and can be specified by a local path (e.g., `/Users/p/myrules`) or a path starting from the project root (e.g. `project://myrules`). Local rule sets are editable which means they can also be enabled on a rule-by-rule basis. It is advised to store project and team specific rules as part of the project. This means the rules are always available, up-to-date with the code and following the same flow as regular code (e.g., branch, review, merge). When rules follow the same flow as the code, new APIs and the rules needed to use them properly can be added to the project and reviewed as a whole.

Previous research and experience have shown that customization at project level is the most successful. This provides the needed flexibility to tailor the enforced coding guidelines to the code, but also ensures that the team has a joint approach to how the code for a project should be developed [15]. Individually customized rules might lead to disagreements, while company-wide rules might be too general to be easily applied.

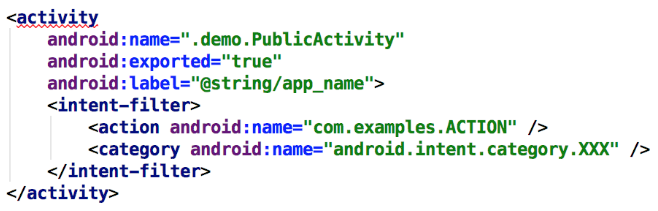
It is possible for a rule to disable another rule. This feature can be used to improve imposed rules. For example, when a rule is imposed on the developer but it is not fully applicable to his project, e.g., because it requires too many manual adaptations, it is possible to create a replacement rule. This replacement rule can be distributed to one team or project and disables the original rule when it is active. When a remote rule is disabled or replaced, the author of this rule should be notified. It is possible that it is a generally applicable improvement and the rule can be updated accordingly for other teams or projects that use it in a remote rule set.

The discussed features to disable rules have been designed to improve the usability of the tool for developers. They are in line with the philosophy that developers' productivity benefits from their ability to customize their development environment for their preferences, and to give them a significant amount of freedom in that regard. In that philosophy, it is preferable to have developers disable some rules rather then uninstalling or neglecting the tool completely. Importantly, this does not necessarily result in guideline violations slipping below the radar, since security and management can still have these rules enabled in later phases of development, as will later be explained in Section 5.

In contrast to our philosophy, we have noticed in practice that management does not necessarily care about the usability for the developers and that they often prefer to impose certain rules regardless, including the use of the Sensei plugin. However, in sections 4.1 and 6.1 of this paper we provide an argumentation and evidence that usability is an important metric for effective continued use of software security tools.

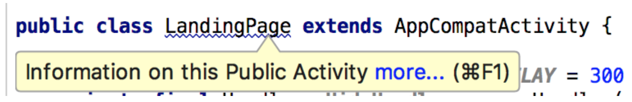
3.4. Explaining Rules

In order to mark violated guidelines, our plugins make use of existing IDE features to flag coding mistakes. In most IDEs the code markings by default have three levels of severity: error, warning, and information. We recommend to mark coding guideline violations as errors. Traditional error-level markings are usually immediately addressed by the developer, while warning-level markings are more frequently ignored [13]. This is the case because error-level warnings in an IDE typically indicate a problem in the code that will result in a compilation failure. Currently error markings by our tool still allow successful compilation of a project, but several clients have requested the markings to result in compilation failures, equivalent to errors marked by the IDE itself. This is not surprising, as it is in line with the default behavior of popular IDEs such as Visual Studio. When



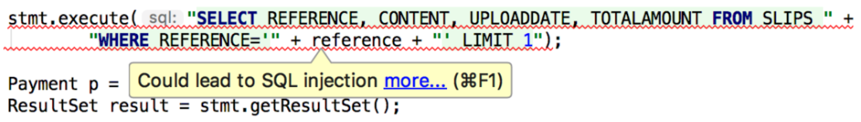
```
<activity
    android:name=".demo.PublicActivity"
    android:exported="true"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="com.examples.ACTION" />
        <category android:name="android.intent.category.XXX" />
    </intent-filter>
</activity>
```

Figure 9. Error marking on an XML opening tag.



```
public class LandingPage extends AppCompatActivity {
 Information on this Public Activity more... (⌘F1) LAY = 300
```

Figure 10. Information marking on a class name definition.



```
stmt.execute( sql: "SELECT REFERENCE, CONTENT, UPLOADDATE, TOTALAMOUNT FROM SLIPS " +
        "WHERE REFERENCE='" + reference + "' LIMIT 1");

Payment p =  Could lead to SQL injection more... (⌘F1)
ResultSet result = stmt.getResultSet();
```

Figure 11. Short description when hovering over an error.

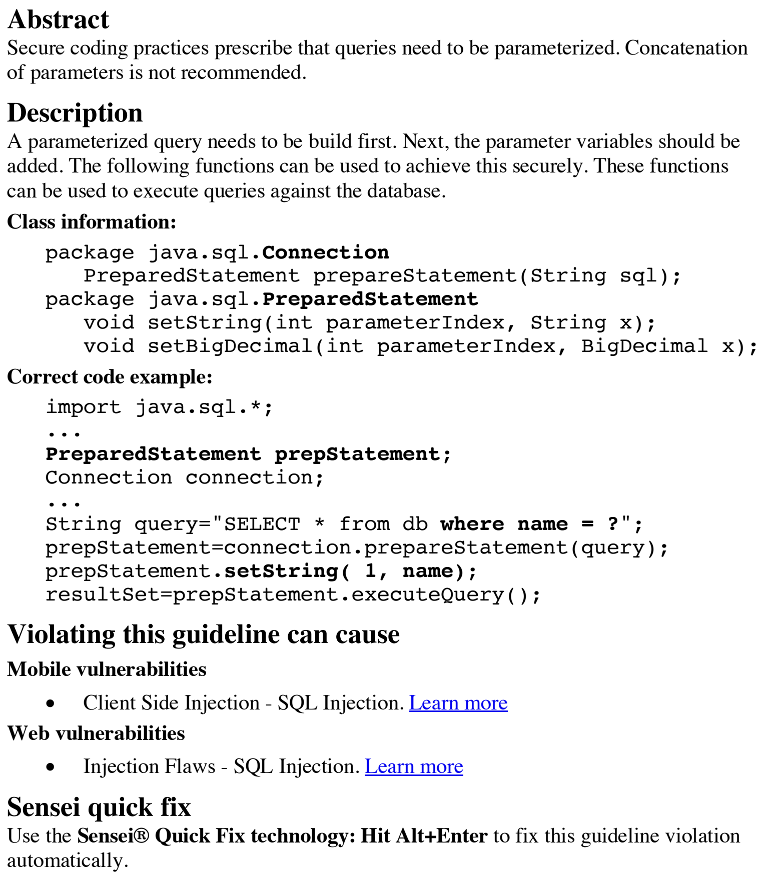
Visual Studio's C compiler compiles code that uses the `sprintf` function, it throws a compilation error warning the developer that the function may be unsafe.

An example can be seen in Figure 9 where the opening `<activity>` tag is marked as an error. This marking makes the code fragment stand out and attracts the developer's attention. In particular in Extensible Markup Language (XML) code we only mark the opening tag, and not the entire XML tag and its content. This would overwhelm the developer, and it would not be clear which part of the code is lacking.

Permanent markings that remind developers of security implications of their decisions should be marked as information. To continue on the example of private and public activities, in the code file that implements the activity, we mark the class definition at the information level. Hovering over the marking informs the developer whether the activity is configured as public or private, and provides a direct link to detailed information about the security implications. This marking is shown in Figure 10. Note how the markings are clearly visible and noticeable, but at the same time non-intrusive to developers already used to their IDE flagging code fragments.

The marking of code is accompanied by three different descriptions. This information is important to ensure the continued use of the tool [13, 11]. Developers build trust with analysis tools, and this trust is quickly lost if they do not understand the tool's output [20]. The first description is the short error description, i.e., the text that appears when the developer hovers their mouse pointer over the marked code. It should be just one line. The purpose of this description is to attract attention, inform the developer that something should be addressed, not to explain how to address it.

We have learned through user feedback that it is most effective to attract the user's attention by starting with the "why" [21], the consequences of not addressing the issue. The short description should always indicate the possible vulnerability class. An example is the "Could lead to SQL injection", as shown in the screen shot in Figure 11. Despite our emphasis to use secure coding guidelines and not to search for vulnerabilities in the previous sections, we still present the vulnerability class here. Starting with the potential consequences makes the developers realize the severity of their mistake and encourages them to immediately address it. In the additional descriptions used to explain how to resolve the issue, the vulnerabilities themselves are secondary. The focus when resolving the problem is still on the solution rather than the vulnerability to avoid needlessly complicating the issue.



Abstract

Secure coding practices prescribe that queries need to be parameterized. Concatenation of parameters is not recommended.

Description

A parameterized query needs to be build first. Next, the parameter variables should be added. The following functions can be used to achieve this securely. These functions can be used to execute queries against the database.

Class information:

```
package java.sql.Connection
   PreparedStatement prepareStatement(String sql);
package java.sql.PreparedStatement
   void setString(int parameterIndex, String x);
   void setBigDecimal(int parameterIndex, BigDecimal x);
```

Correct code example:

```
import java.sql.*;
...
PreparedStatement prepStatement;
Connection connection;
...
String query="SELECT * from db where name = ?";
prepStatement=connection.prepareStatement(query);
prepStatement.setString( 1, name);
resultSet=prepStatement.executeQuery();
```

Violating this guideline can cause

Mobile vulnerabilities

- Client Side Injection - SQL Injection. Learn more

Web vulnerabilities

- Injection Flaws - SQL Injection. Learn more

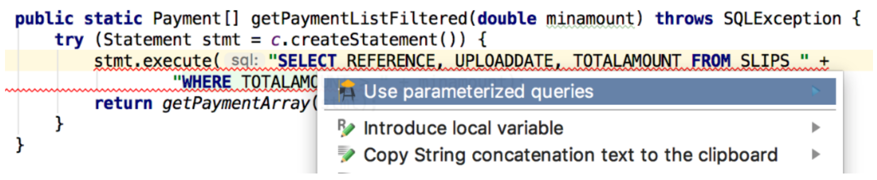
Sensei quick fix

Use the **Sensei® Quick Fix technology: Hit Alt+Enter** to fix this guideline violation automatically.

Figure 12. Example of the full coding guideline.

Next to the short description a "more..." link is created by the IDE for the interested developer. Upon clicking this link a pop-up is opened to show a more elaborate Hypertext Markup Language (HTML) page. This is the second description. Figure 12 shows an example. This description is called the full coding guideline. The page starts with a short abstract, stating in one sentence what should be done, such as "Secure coding practices prescribe that queries need to be parameterized". The page's next section presents in detail what it means to use parameterized queries and gives an overview of the approved API methods. A small secure code example is included as well, since previous research has shown examples are the fastest way for developers to understand the problem [13]. There is no mention of vulnerabilities or exploits in this description until at the end. The reason is that we do not want to disturb the developer during development, as this is one of the main reasons security tools fall out of use [18, 11]. They should instead quickly find out how to comply to the coding guideline without spending much time or effort. The last section of the description contains a list of possible consequences when the developer fails to address this issue. Each item in the list contains a link to the Secure Code Warrior training platform to learn more about the vulnerabilities and how they are exploited. This way an interested developer (with too much time on his hands?) can still easily find the necessary information to learn the details of each vulnerability and the possible attacks.

The third description is visible to the developer when they press the IDE's key combination to start resolving the issue. A drop down menu appears containing the possible quick fixes' descriptions and also some options to disable this type of inspection locally or globally. Figure 13 shows an example. In this menu we provide a very short description of the actions that will be performed when this code transformation is chosen, such as "Use parameterized queries". A brief yet descriptive quick fix description allows developers to decide quickly whether the fix is appropriate for them. If the effects of applying the quick fix are well understood, the developer will trust the tool and apply the



```
public static Payment[] getPaymentListFiltered(double minamount) throws SQLException {
    try (Statement stmt = c.createStatement()) {
        stmt.execute( sql: "SELECT REFERENCE, UPLOADDATE, TOTALAMOUNT FROM SLIPS " +
                "WHERE TOTALAMC
        return getPaymentArray(
    }
}
```

Use parameterized queries
Introduce local variable
Copy String concatenation text to the clipboard

Figure 13. Example of the quick fix description.

fixes more often. Sometimes the developer needs to choose between different possible solutions. However, it is advised to keep the number of fixes as low as possible, as to not complicate the issue unnecessarily.

4. RULE FEATURES

This section describes advanced features in the rules that have been developed following user feedback. We explain the problems they tackle and provide example rules for each of the features.

4.1. Usability: Lowering Effective False Positives

It is important to choose the right error level for the developer to pay attention to the markings, but also not to overwhelm them with markings to the point that they start to ignore them. Since the rules can be created by anyone in the team, they should not be too obtrusive. To a rule writer, a false positive is an incorrect marking of code that is not violating a coding guideline. However, to a developer, a false positive is any code marking that they do not intend to fix and ignore instead [22]. A false positive from the perspective of the developer is also called an effective false positive [15] (EFP). To ensure the usability of the tool, the EFP rate should be sufficiently low [15, 18].

To demonstrate EFPs we take a look at Operating System (OS) Command injection. One of the OWASP ESAPI Banned APIs is `Runtime.exec`. This API is used to execute OS commands in Java programs. When user input is added to this command, OS command injection is possible and the attacker can gain access to the underlying OS. Using this information it is possible to create a rule that marks all uses of the `Runtime.exec` method. While this is a good coding guideline, a security conscious developer recognizes that the method needs user input before it can lead to OS command injection. In rare occasions an OS command might be necessary for functionality and perfectly valid and secure. For example, launching another software product can be done securely as long as the command is hard-coded. An example of insecure usage of the `Runtime.exec` method can be found in Listing 1, examples of secure usage in Listings 2 and 3. The two secure examples are still violations of the above coding guideline, and they get flagged, but an experienced developer has no intent to fix them, meaning they are two cases of EFPs. A similar example leading to EFPs is when hard-coded SQL queries are created without using parameterized queries.

In order to keep the EFP rate sufficiently low we have introduced the concept of trusted input. Hard-coded input is automatically trusted, since a user can not influence it. However, function parameters and variables from other origins are considered untrusted by default. This is still in line with the philosophy to protect methods from future use, where we want to flag violations that can one day lead to vulnerabilities. The requirement in rules of untrusted input can be added to arguments, this can be done using the YAML syntax or by using the GUI as shown in Figure 14. The next step is to define trusted sources. This also avoids the EFP in Listing 3. The resulting rule can be seen in Listing 4.

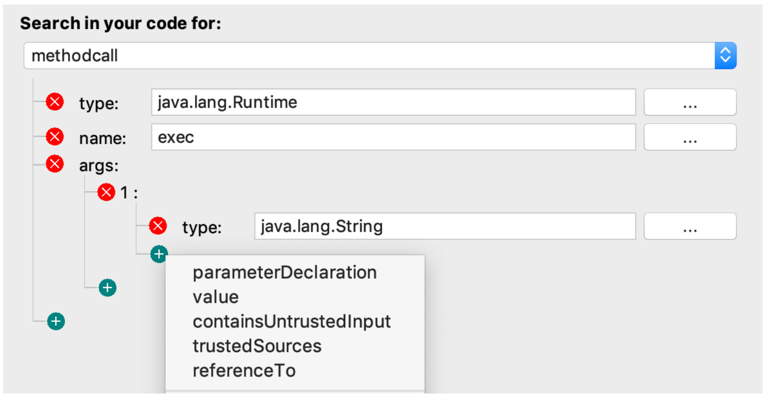

Search in your code for:
methodcall
type: java.lang.Runtime
name: exec
args:
1 :
type: java.lang.String

Figure 14. Graphical user interface used to add an extra constraint to the arguments of a methodcall.

```java
public void executeCommand(String command){
    Runtime r = Runtime.getRuntime();
    r.exec(command);
}
```

Listing 1: Insecure usage of Runtime.exec

```java
public void executeCommand(){
    Runtime r = Runtime.getRuntime();
    r.exec("explorer.exe");
}
```

Listing 2: First secure usage of Runtime.exec.

```java
public void executeCommand(){
    Runtime r = Runtime.getRuntime();
    String command = getSafeCommand();
    r.exec(command);
}
```

Listing 3: Second secure usage of Runtime.exec.

```yaml
search:
 methodcall:
  name: "exec"
  type: "java.lang.Runtime"
  args:
   1:
    type: "java.lang.String"
    containsUntrustedInput: true
    trustedSources:
    - methodcall:
       name: "getSafeCommand"
```

Listing 4: Rule trigger to prevent OS command injection.

Another way to allow rule writers to create more targeted rules and to keep the EFP rate low, is to provide trigger scopes. Trigger scopes can be added by using the "in" keyword in the YAML syntax. The "in"-clause can define restraints on the context. This makes it possible to create a rule that prevents the usage of `Runtime.exec` except in in a class `AppLauncher`. Scopes like this can also help with performance, i.e., meeting the real-time checking requirement, since rules that are out of scope can be skipped during analysis.

In the old rule editor rule scopes were added to the rule by selecting the type of scope and filling in some fields. By migrating rule scopes to the YAML syntax the scoping of rules becomes much more flexible. The rule scopes that can be migrated to trigger scopes in the YAML syntax are the

```java
Cookie myCookie = new Cookie("secure", "success");
response.addCookie(myCookie);
```

Listing 5: Insecurely configured cookie.

```java
Cookie myCookie = new Cookie("secure", "success");
myCookie.setSecure(true);
myCookie.setHttpOnly(true);
myCookie.setDomain("sub.domain.scw.com");
myCookie.setPath("more/narrow/path");
response.addCookie(myCookie);
```

Listing 6: Securely configured cookie with library usage.

```java
public void safeAddCookie(Cookie myCookie, HttpServletResponse response){
    myCookie.setSecure(true);
    myCookie.setHttpOnly(true);
    myCookie.setDomain("sub.domain.scw.com");
    myCookie.setPath("more/narrow/path");
    response.addCookie(myCookie);
}
```

Listing 7: Library replacement API

```java
Cookie myCookie = new Cookie("secure", "success");
safeAddCookie(myCookie, response);
```

Listing 8: Securely configured cookie with library adoption.

class scope, method scope, file scope, Android context scope, and Android build property scope. Descriptions of these scopes can be found in Appendix A.

There are two more rule scopes, for which migration to trigger scopes is not useful. The project scope allows us to enable or disable rules based on the name of the project. This is useful when different rule sets are required for each project in a company. This scope is no longer needed since we now allow rule sets to be stored in the project, which is a lot more convenient then adding a scope to each rule separately. The library scope can be used to enable rules based on the presence of a library. This way we can disable a rule if the fix uses a library that is not used in the project. Since this scope is created for the fix of the rule and not the trigger, it cannot be added to the YAML syntax and remains a property of the entire rule.

4.2. *Support for Libraries*

Often quick fixes are small code changes, such as adding a preceding method call or changing a parameter, but sometimes they involve more elaborate pieces of code. An example for this is adding a cookie to an HyperText Transfer Protocol (HTTP) request. Before adding the cookie, it needs to be properly configured. The insecure and secure code are shown in Listings 5 and 6, respectively.

If this fix is applied at multiple locations in a project, it can result in code bloat. It might then be better for the company to provide a method that replaces the original `addCookie` method and that configures the cookies automatically, such as the example of Listing 7. The new rule for cookies is then to replace the `addCookie` method with the `safeAddCookie`, as shown in Listing 8.

The first example, where the cookie is configured properly, is a library usage rule. This type of rules provide guidance on using a library correctly. The trigger of the rule is on APIs from the library. Code fragments are refactored without involving additional libraries, only libraries of which we are sure are used in the trigger.

The second example, where the insecure code is replaced with API calls from a different library, is a library adoption rule. Instead of providing guidance on the correct usage of the APIs, such rules

```
1  <application
2          android:label="@string/app_name"
3          android:usesCleartextTraffic="false">
```

Listing 9: Snippet from an Android manifest file.

promote the adoption of a new library. These rules have a trigger in one library but their fix uses another library.

As a proof of concept for library adoption rules, support has been developed for the OWASP ESAPI library. Among others, the OWASP ESAPI contains replacement methods for commonly used insecure Java Development Kit (JDK) methods, the so-called OWASP ESAPI banned APIs**. To support the OWASP ESAPI in companies that adapt it, a rule set was created to enforce the replacement of the banned APIs with their alternatives from the OWASP ESAPI. Feedback from these companies showed that this set of library adoption rules was intuitive and easy to use for developers. Importantly, it increased the speed of the library's adoption.

Library usage rules are generally applicable to code bases because the trigger and fix of these rules use APIs from the same library, thus ensuring that the rules never mark any code when the fix is unavailable. The trigger and fix for library adoption rules depend on different libraries. This implies that an applied quick fix can result in the use of unavailable APIs.

We have already introduced library scopes that can be used for this purpose. When the library used in the quick fix serves as a scope of a rule, it will not mark any code if the library is unavailable.

4.3. Support for Detecting Design Flaws

As discussed before, coding guidelines are enforced through mostly local analyses. This allows us to intervene earlier in the development process and makes it possible to perform the analyses in real time while the developer is typing. For this reason the focus of the approach is mostly on implementation bugs. Detecting design flaws in the application code (rather than in the APIs it relies on) is typically much harder. Still, we have learned that various flaws can be tackled through the use of configuration files and the previously described trigger and rule scopes.

An example of a flaw that is difficult to detect with local analyses is excessive security logging. It is crucial to log important security events, but too much logging can make it difficult or impossible to locate certain events. While enforcing guidelines can help with logging securely (e.g., not logging sensitive data, not logging unsanitized input, logging in proper format) it is difficult to monitor the frequency of the logging through local analyses. Other examples are the use of transport layer security, or whether authorization is needed or not.

One scenario that to the contrary enables us to detect some design flaws, is when popular frameworks are used to handle security features. For example, enforcing the use of transport layer security in an Android app is as simple as adding a line to the Android manifest about clear text traffic. This can be seen in Listing 9, line 3.

Another example is the use of encryption. It is trivial to detect if a deprecated encryption algorithm is used by means of a known API, but it is much harder to detect the absence of encryption through local analyses. One interesting class of mistakes that we observed was developers XOR'ing data, or encoding data, rather than encrypting it. As a solution, one client created a coding guideline that requires functions whose name contains "encrypt" to perform encryption through some of their approved API methods. If such a function only performs encoding or XOR operations, it implies that the required API calls are missing. In that case the tool suggests quick fixes that insert the necessary API calls. In this case, the quick fixes are only partial fixes: They inject code that invokes encryption routines on an unspecified string or byte array. It is then still up to the developer to remove the XOR operations and fill in the correct string or array identifier. This emphasizes again why it is beneficial to provide fixes in the IDE during development time. When this rule is used during the development

**`https://www.owasp.org/index.php/ESAPI_Secure_Coding_Guideline#Banned_APIs` of a new method, the developer starts by creating a function declaration. When a function exists with "encrypt" in the name and an empty body, this is marked, as the required API calls are missing. The fix then inserts the required API calls, leading to comfortable and intuitive security help for the developer.

Finally, we have improved context-awareness to detect flaws by adding more rule scopes. One such example is the Android context scope. As explained earlier, in the Android manifest a developer can configure capabilities of components such as activities and broadcast receivers regarding their communication towards the OS. They can listen to any other application, or only to authorized applications, or only to the own application. The Android context scope allows us to enable rules based on the configuration of the relevant component, so that we can enforce different rules for different levels of exposure. Our tool can for example allow communication of sensitive information between classes that are configured as private components, but not between other classes.

4.4. Testing Rules

Testing custom rules is a challenge. When new rules are created, the rule writers first have to test the behaviour of the rule manually. They develop a few code fragments they expect to be marked, as well as some that should not be marked by the rule. They then applies the transformations and manually inspect the resulting changes. The rule wizard helps speed up these tasks by providing preview panels during rule creation. A custom rule writer in a company, however, cannot be asked to perform the manual checks again every time they install updates to our plugin (including its underlying analyses) or to the IDE itself. Such updates always risk altering the outcome of a rule. Instead automated unit testing is needed.

The plugin developer has sufficient capabilities to automate unit tests to verify the behavior of the analyses. To do this they also create a few demo rules and tests the behavior of these rules. Since they have access to the code of the plugin, they can simply write unit tests and (directly) call internal plugin and exported IDE methods to test the markings and transformations and to compare them to the expected results. In other words, they can write code snippets that check automatically whether or not updates to tools alter the deployment of existing rules. Such testing is unavailable to the custom rule writers in a company, however, which do not have access to —and definitely do not want to learn— the internal plugin APIs.

A better testing framework is hence required, such that the rule writers in companies can indicate the expected outcomes of every rule they wrote on a number of code samples, and such that they can test entire rule sets automatically before updating any component in the IDE or plugin.

If we allow rule writers to define tests in the plugin itself, and store them in the rule sets, these tests can automatically be performed when loading a new rule set and after every IDE and plugin update. We can then notify the user when one of the rules is no longer working as expected.

In the new rule wizard it can be trivial for the rule writer to select one or more of the examples that are marked in the preview panels and allow these examples to be used as the rule tests. This way the rule writers are creating tests for their rules with nearly no additional cognitive effort. At this point in time, implementing the necessary support for rule writer defined tests remains future work.

4.5. Code Generators

When developers use the plugin for a while they get used to certain fixes being available. They know that if they write an insecure SQL query, the quick fix is able to secure it for them. This results in developers purposefully making mistakes because it takes less effort to let the plugin fix it. In order to improve this process we have created code generators. Generators are secure code fragments that can be inserted with a shortcut. Code generators are developed by the rule writer, and rolled out to the developer, just like the rules. They consist of a name and the inserted code. In order for generators to be successful, they need to adapt the generated code to the context, similarly to how the quick fixes adapt when they are applied (relying on the use of the template language in their specification). From our experience with early versions of the generators, where the inserted code was not adapted to the context, e.g., to reuse the variable names occurring in the original code, we learned that that such a lack of adaptivity was a blocker for the take up by developers. More adaptive generators have not been implemented so far, validating whether this will improve their use remains future work.

5. THE PLUGIN AS A MANAGEMENT TOOL

Coding guidelines can provide a good measure for security in a software product. Where vulnerability scanning can only provide an indication of the vulnerability density, they do not provide the full picture. In the case, e.g., where a large number of SQL injections are found, this could indicate poor database security. But it can also mean that there simply are a lot of database queries, with many of them done securely. For coding guidelines a relative measure can be designed, by comparing the number of guideline violations to the number of times the code complies to guidelines. Since complying to strong coding guidelines leads to secure code [14, 23], we get a better indication of the security in the software product.

While the plugin is useful as a tool to aid the developer, the option to measure guideline deployment also hints at its potential as a management tool. It is possible to track the changes developers makes to projects and log the guideline violations that they introduce and fix, with or without aid of the quick fixes. This makes the return on investment clear to companies using the tool.[††] It can be used to track the performance of individual developers, and then to give more targeted and individually tailored training.

While individual performance can hence be measured and improved, with developers working in different branches, and hence different states of the project, it is hard to get a good overview that way. To resolve this, we also give managers the possibility to use the plugin technology as a headless scan that can be performed outside of the IDE. The same analyses can then be automatically performed by integrating them in the CI/CD tools. A manager dashboard, hosted on a web app, helps the managers to interpret this data.

6. EVALUATION

6.1. *Usability*

For the installation of the tool, existing IDE features are used. The "Plugins" menu is well-known to most developers. We have monitored this process with more than a hundred developers, both well experienced and students, and it only takes several minutes. The startup time of the IDE is not measurably affected by the tool. The tool only performs a license check, of which the duration is shorter than the measured variation in IDE start-up time.

When analyzing newly written code, the longest time we have measured that is needed for the analyses to finish is 29 ms. This is far below the threshold of 125ms to be considered real-time.

This means that a developer is not hindered during development unless they are violating a coding guideline. When code is marked it is possible that a developer spends extra time understanding the issue and trying to fix it. Even though the initial development time may increase, the total time spent to secure code is lowered due to early intervention. However, it is still beneficial to minimize the additional time and effort spent by a developer addressing marked code. While interviewing developers, anecdotal evidence has shown that they quickly show trust in the rules and spend little time trying to identify false positives. They quickly determine if the available quick fix is applicable in their code and move on.

In one experiment, that we describe in more detail in Appendix B, we have analyzed the usage statistics of 32 users completing a programming exercise. Out of the 32 users, 15 received active

[††]In empirical experiments that we are currently designing, we plan to use this to track the effectiveness of the plugin. We plan to compare a control group of developers to a test group. The control group will be monitored, but for them the code markings and quick fixes will be disabled. The test group will have access to the full functionality of the tool.

help from the tool, the others subjects were monitored only. During this test 98.4% of code markings shown by Sensei (n = 247) were resolved by the developer. Out of the resolved code markings 73.3% were fixed using the quick fixes. All of the users (n = 15) used at least one quickfix, with an average of 12.71 (s = 4.73) quickfixes used. The remaining code markings have been removed manually, either by fixing the violation or by removing the violating code entirely. This is a high level of engagement, compared to the lower than 20% "Apply fix" rate reported by the code review tool Tricorder [15]. On average the subjects resolved the issue within 19.10 s (s = 25.22 s) of writing the violated API call. This means that developers are spending comparatively little time understanding the issues and applying fixes. By comparison, for Tricorder and SpotBugs the time between writing the violating code and fixing is usually several days [15, 24].

During our experiment the use of Sensei on average increased the total development time with 11.72%. This is a relatively low increase considering the programming assignment was to complete security critical features and hence the subjects were frequently confronted with feedback from the tool. It is also important to note that for all of the subjects the experiment was the first time they were making use of the tool.

In the experiment with Sensei, only 1.6% of code markings were ignored, which is a low EFP rate. After carefully improving their analyzers, Tricorder reached an EFP rate of around 5%. When using SpotBugs, research reported that 58% of the reported issues were never reviewed. Out of the reviewed bugs 55% were eventually fixed [24]. We observe a big gap in EFP rates with Sensei and Tricorder having a rate of 5% or lower on one hand, and SpotBugs having 77% on the other hand. The reason for this is the customization of rules. Tricorder allows creating new analyzers and their quality is closely monitored. For Sensei, developers are given carefully tailored rules, often written by the engineers themselves, and relevant to their project. These efforts are clearly greatly improving the usability of the tool and hence the trust of the developers.

In total four guideline violations were ignored and not addressed by the subjects. All four were violations of unique guidelines, and so there was no guideline more often ignored then others. Of the ignored guideline violations was one ate risk for resulting in SQL injection, one for resulting in path traversal, and two of them were related to issues regarding file upload vulnerabilities. However, it can still be useful to monitor if specific rules are leading to EFPs more often than others, as this can indicate a rule of poor quality. The IDE has a feature to suppress code warnings locally or globally. If we keep track of how many times a rule is suppressed and how widely, this can help us identify rules that are often leading to EFPs. In the rule editor it is also possible to disable or replace local rules. These are similar changes that need to be tracked. We can then alert the rule writer that his rule is often suppressed or disabled. We can even automatically disable such rules for all developers. Tricorder has a similar feature. When an analyzer in Tricorder is ignored often, it is disabled for all users and its author is given an opportunity to improve it before it is re-enabled.

6.2. Rollout Best Practices

Getting the plugin rolled out within organizations is tricky. For solutions that developers did not ask for, and especially security solutions, one gets exactly one shot to get it right. While the tool is a developer tool, it is also a management tool, and it is usually purchased by management. Which is why it is important to show value for both user groups.

Management is sometimes hesitant to purchase and roll out Sensei in their teams without any starting rules. They see it as buying an empty box that requires a large effort to fill by developers and security experts. At the same time the flexibility and customizability of the enforced rules is what makes Sensei powerful at eliminating security problems early in the SDLC. Many rules are not applicable to all code bases and fixes are often project specific. One solution could be to provide rule sets for commonly used frameworks, such as Android and Spring, as in these frameworks the fixes are well known and commonly used. Alternatively, we are designing a tool that automatically suggests rules based on the project. This tool will identify secure use of APIs and will generate rules that enforce the identified API use. The rule writer can then accept or edit some of these rules to create an initial rule set. Previous research has been performed to identify crypto API rules from code changes [25]. Efforts have also been made to automatically generate patches from code repositories and their histories, using different algorithms [26], including learned from human-written patches [27] or correct code [28]. While this research tries to automatically patch bugs, the approaches can also be used to create rules to apply the discovered patches more broadly and to do so during the writing of code rather than afterwards.

From experience, we learned that ideally our plugin is rolled out when new rules do not mark any existing code. This is when a project kicks off and zero lines of code have been written. Alternatively it can be rolled out when a new API or library is introduced in the project and rules will be written for this library or API. Few projects are developed from scratch, however, so the reality is that the plugin needs to work in an already developed product. In that case, rolling the plugin out with all rules switched on can be overwhelming to developers, as they are presented with a huge number of violations. In addition, developers are often hesitant to fix issues they did not introduce in the code themselves, and they might not even have permission to change code that is not theirs. This results in a large number of EFPs, which we want to avoid.

When developers create their own rules from scratch they are working on a certain branch of the project. They usually create very targeted rules to fix or enforce small things in the project files they are working on. When they create the rule they inspect the violations and fix the markings. The rule and fixed code are pushed to the code base simultaneously. This typically leads to few EFPs. However, often the application security team of the company imposes rules as well. At one point, we had a client where the security expert created a large number of rules and imposed them onto the developers without fixing any of the resulting violations. It did not come as a surprise that this resulted in a great number of EFPs and out of the 20 developers that had the tool installed, all of them had disabled its markings. To avoid such failures, we recommend two approaches to keep the EFP rate low for imposed rule sets.

Firstly, in the ideal scenario the person responsible for security or the application security (appsec) team creates a number of rules and looks at their violations in the code to inspect their severity. When rules result in few violations, the appsec team can safely roll out the rules without resulting in too many EFPs.

The roll-out is more challenging when a rule results in a great number of violations that are not trivially resolved. In that case the appsec team should create a developer task force. Their task is to create APIs to resolve the rule hits. They then turn the appsec rule into a library adoption rule and fix all marked code with this rule. In the process of doing so, many corner cases can be encountered to can help to fine-tune both the API and the rule. The new API, the rule, and all code fixes can be pushed to the code base simultaneously.

The ideal scenario might not apply in practice, however. It is possible that the code base is simply too large to start fixing all code markings. We have had clients where a strong rule resulted in over 3000 violations. It can also be the case that when the appsec team creates a task force, this developer time is payed by the security budget, not the development budget. In such cases it is not beneficial to spend developer time to fix the existing issues in the code before rolling out the rule.

We then instead recommend the second approach, in which the rules are rolled out company-wide without fixing the code markings. In order to keep the EFP rate sufficiently low, the violations are only shown partly. For example, it is possible to only mark code that the developer is currently working on. This way the violations are gradually shown (and resolved) without resulting in an overly large EFP rate. We are currently designing a feature that should make this possible.

Related tools will be described in more detail in Section 7. Most of those tools operate in the test phase of the SDLC; after completion of a feature a scan is performed. In the scan results usability issues are less critical, as they are not markings that can disturb a developer during development. Instead it is often a security expert who will analyze and prioritize the results of security scans by placing them into the bug tracking system. Common features for those tools are integrations with common bug tracking systems to allow them to publish bugs automatically.

We observed that many tools provide functionality to disable certain rule reports through configuring security policies. This is a necessary feature to remove or hide classic false positives. However, this disabling of reports is designed to help security experts keep a good overview of the application state and to help prioritize more severe issues. With the exception of the Fortify Security

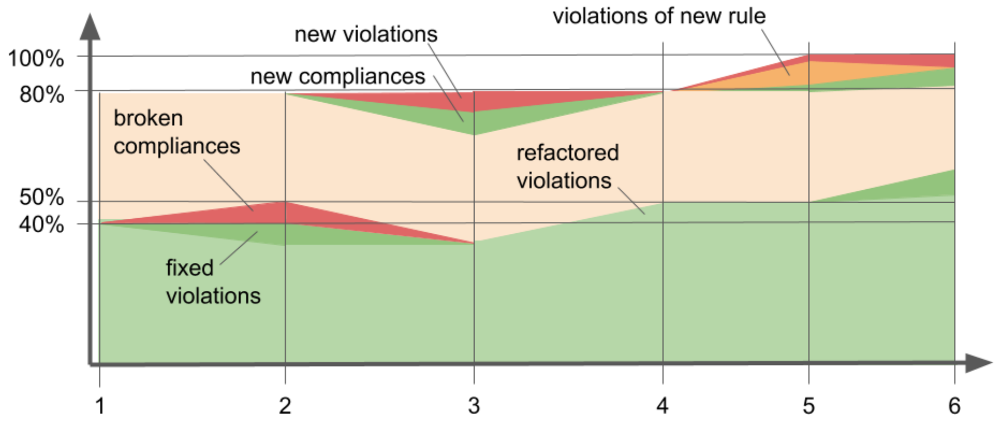


Figure 15. An example project time series on the manager dashboard, showing the progress over the duration of 6 scans.

Assistant that disables rules to speed up the scans, disabling rules themselves is rarely supported as needed to improve the usability of the developers.

6.3. *Rule creation*

When testing the rule wizard using the YAML syntax we observed a significant speed-up in writing rules. Even for users that were experienced with the old rule models, the preview panels in the rule wizard and the context-aware suggestions greatly improve the rule-writing process.

The majority of comparable tools allow writing custom rules or analyses in some way or another. Writing rules for them is done through complex but well documented APIs (SpotBugs, Tricorder, Checkmarx) or more user friendly custom formats such as XML (Fortify, SecureAssist). Of these, Checkmarx is the only one that provides a Query editor and manager, albeit not built into their static analysis tool. For all other tools, the rule writers can use an IDE or text editor of their choice. None of the tools allow creating rules on-the-fly in the code. Sensei does allow this, which greatly improves the speed and usability of writing rules, as context-aware suggestions are made, but also improves testing of the rules since their markings can be observed live as the rule is being created.

6.4. *Project Management*

Our approach and the collected information described in Section 5 allow interesting compliance visualization to support management. Figure 15 shows a potential visualization of the compliance history. Notice that it differentiates more than the number of compliant vs. non-compliant cases.

The first category is when some of the violating instances have been fixed or some of the compliant instances have been broken. We mark the fixed violations in a darker green and the broken compliances in red. In Figure 15, the first scan results showed half of the currently known instances were compliant, as half of the graph starts as orange and the other half light green. In the second scan an equal number of compliances were broken as violations were fixed. With only the compliant and violating parts, no changes would be visible in such data, since the compliance rate did not change. It is clear from the dark green and red parts, however, that some instances have been changed.

When new code is written it is also important to distinguish this from the previously described code rewrites. For this reason we also mark new compliances and new new violations. New instances are in red and dark green, in this case on top of the graph. This can be seen in Figure 15 in the third scan. No existing code was changed in this scan and newly developed code contained an equal number of compliances and violations.

It is possible that during refactoring some instances are deleted. In this case no new compliances or violations are introduced, but the compliance rate still changes. This can be seen in Figure 15 in the fourth scan, no red or dark green is present in this scan, but the compliance ratio still changes.

Finally, it is possible that new rules are added at some point in time. These rules then cause new violations or compliances to be found, but they do not represent new changes to the code. It is possible that these issues have been there since the first scan but they remained undetected. To reflect changes caused by new rules we create an unknown section. It is added retrospectively to all scans previous to the scan that introduces the new rule. The category is visualized as transparent and its value is the sum of the compliant and violating instances of the new rule in the scan that introduces it. This can be seen in Figure 15, in the first four scans only 80% of the graph is colored, this means that 20% of the current instances have only been discovered with new rules in the fifth scan. Violations of the new rule are marked in orange, not in red since red is used for new violations, these are not necessary new violations, only newly discovered.

Since the manager dashboard is still in development we have not enough user feedback yet on whether or not these are good metrics and visualizations for the security state of a software project.

Some of the comparable tools do not provide any management dashboards at all (Tricorder, SpotBugs, SecureAssist, Snyk, ASIDE). We believe that in order to convince managements and to help them roll out the tool, it is important to show a return on investment. Other static analysis solutions (Fortify, Veracode, Checkmarx) do provide management dashboards. The most common features in these dashboards are tracking the amount of vulnerabilities, the severity of different vulnerabilities, and the vulnerability categories. In comparison to our proposed time series they do not focus on verifying the human capital. In contrast, we believe that measurements like the ratio of new compliances to new violations are important to track the developers' skills and awareness. The dashboards of other tools are instead mostly designed to monitor the application state.

6.5. Codebase Impact

The usability measurements presented so far suggest that when Sensei is used, the secure coding guidelines are applied most of the time. During a trial of the plugin, described in more detail in Appendix C, one client has tracked closely whether or not the enforced guidelines actually prevented the introduction of vulnerabilities early on. The trial was done with five developers for the duration of three months. They reported a total of over 200 confirmed bugs being prevented. The most common issues involved sensitive information leakage and tapjacking vulnerabilities in their mobile application.

A limitation of the tool's local analyses is that they do not allow us to detect whether or not a certain input has already been sanitized before flowing into the routine being analyzed. This is in line with our approach and goal of enforcing coding guidelines that defend every routine for future use, i.e., such that it is still secure whenever it might be reused with unsanitized data. So if the local analyses identify a lack of local sanitization, the developer will be expected to let the routine sanitize that input again. At first sight, this might result in the same data being sanitized multiple times within an application, which will negatively impact performance.

In practice, however, this proves to be largely a non-issue. In practice, APIs are not designed in a vacuum. Instead they are developed with potential application architectures in mind. Furthermore, when concrete applications are first designed, security and application architects also take into account best practices for secure architectures (that is, if they care for security-by-design). Similarly, the coding guidelines can be co-designed with certain application architectures in mind. Doing so provides an easy mitigation of the potential issue of redundant, multiple sanitizations.

For example, consider the case of cross-site scripting (XSS) attacks. It is a common misconception that in order to prevent stored XSS attacks, user input should be encoded before it is stored in the database. A better recommendation is to encode the database output when it is used, as the stored data may be used in different contexts, requiring different encoding methods. For example, a string value may be displayed on the HTML page and also used in a JavaScript script on that same page, resulting in two different, but simultaneous escape requirements. We learn that data should always be sanitized before it is stored in the database and encoded before it is displayed in the web or mobile application. Since these are usually the ends of the data flow, no data needs to be sanitized or encoded twice. If the rules are co-designed with the secure application architecture, it becomes trivial to enforce the sanitization and encoding routines at the correct locations in code. The above does not imply that Sensei is the one and only tool that solves all potential software development security issues. To detect issues as early as possible, i.e., in real-time as the developer is writing code, analyses have to be light-weight. This implies that all possible execution paths in the entire program cannot be exhaustively considered, and some types of vulnerabilities, including design flaws, can go undetected. This is a common trade-off, therefore tools that are used early in the SDLC such as Sensei should be complemented with more complete scanning solutions deployed later in the SLDC. An example of this strategy is the Fortify Security Assistant. This IDE plugin is used earlier in the SDLC than other Fortify tools, but only uses a subset of the available rules to improve developer usability. It helps a set of vulnerabilities to be detected earlier, and hence saves money and time fixing those issues, but it does not provide the full protection that, e.g., Fortify on Demand does. In the related work section, we will discuss where we consider Sensei to improve over Fortify Security Assistant as an early SDLC tool.

In conclusion, from our current evidence we observe a beneficial impact on the codebase by using Sensei, where security issues are addressed truly as early as possible in the SDLC. The application of the rules, if designed properly, has minimal impact on performance, and helps improve code quality in many cases. Future experiments similar to the experiment mentioned in Section 6.1 should still be conducted to more thoroughly evaluate the effectiveness of Sensei at preventing the introduction of security bugs.

7. RELATED WORK

7.1. Commercial Offerings

7.1.1. Fortify Micro Focus Fortify is an ecosystem that embeds application security testing into all stages of the development tool chain. Some of the most comparable tools to Sensei in their ecosystem are Fortify Static Code Analyzer (FSCA), Fortify on Demand (FOD), and Fortify Security Assistant (FSA).

As the name suggests, FSCA [29] performs static code analysis on the source code. It can be built in CI/CD tools and has support for 25 programming languages including Java and C#. Scanning takes several minutes and the results can be shown in a web interface or in integrations with many bug tracking systems, ticketing systems, and code repositories. Fortify recommends using their rule sets that cover over 1000 vulnerability categories and more than one million APIs. Creating new rules can be done in their custom XML format in any text editor [30]. Doing so requires reading extensive documentation and learning the proper syntax. They do not provide a rule editor, instead the rule writer can use any preferred text editor. FOD [31] provides similar features to FSCA but through a web portal, Micro Focus calls this Application Security as a Service.

FSA [32] is a plugin for the IDE, currently available for Eclipse and Visual Studio. It allows security scans in the IDE, where the rule set is tuned such that the longest analyses are disabled by default. The scan can take several minutes during which the developer cannot make any code changes. This is still quite long compared to the real-time results of Sensei and might inhibit developers from requesting scans frequently during development.

Besides the FSA, Fortify offers two more IDE plugins. The FOD plugin is available for Eclipse, Visual Studio, and IntelliJ. It allows the developer to request static assessments from FOD and download the results. A similar plugin is available for FSCA on all three IDEs.

FOD allows scan results to be evaluated against custom security policies in order to hide reports of certain vulnerability types. During installation of the FSA in the IDE, a link to a remote Rulepack Update Server is required, this allows the tool to download the necessary rule information to perform scans. It is also possible to use a local copy of the rules when no network connection is available. The FSA also allows enabling and disabling each individual rule in the UI.

Each tool provides detailed descriptions on vulnerabilities, which focus on explaining the vulnerabilities in detail, in part by providing examples of insecure code. In a second tab, FOD provides a description and code examples to resolve the vulnerability. All tools provide links to reference material and to recommended solutions. FOD also provides links to Secure Code Warrior to provide training on a specific vulnerability. The information provided is very similar to the different descriptions of Sensei, but the focus is more on explaining the vulnerability first instead of the solution.

Both FSCA and FOD provide a management dashboard in which the security status (failed or passed security policy) of multiple applications can be seen. The dashboard of FOD provides more detailed information where it is possible to track the amount of issues over time for each severity. It also provides an insight in the vulnerability categories of most prevalent issues. FOD can track the amount of newly introduced problems in each scan. This makes it possible to evaluate the developers' skills over time, on top of tracking the application state. However, in contrast to Sensei, no clear distinction is made between problems detected by new rules and problems detected in newly developed code.

7.1.2. Tricorder Tricorder [15], which has an open-source version Shipshape [33], is a data-driven program analysis platform integrated into the workflow of developers at Google. Tricorder's design philosophy closely resembles that of Sensei where they put developer usability first. Like Sensei, Tricorder allows developers to share their knowledge by enriching the set of available analyses. However, instead of an easy rule editor, analyzers in tricorder may be written in any programming language. The best support is available for C++, Java, Python, and Go. Analyzers can analyze any programming language. Analyzers are encouraged to provide fixes with their analysis results to improve developer usability.

In contrast with Sensei, the results of Tricorder analyzers are shown in a review tool. In this tool the reviewer can choose from several options: Not useful, Please fix, and Apply fix. The last option will directly apply the fix suggested by the analyzer in the review tool. However, empirical observations have shown that few Apply fix clicks occur. The authors hypothesize that many developers choose to fix the code in their own editor rather than use the code review tool for this purpose [15]. This indicates that the Sensei quickfixes are more usable, since they are applied in the IDE.

Tricorder analyzers are deployed company wide if they meet certain criteria regarding their output. They should be easy to understand, result in few false positives, the markings should have potential for significant impact, and they should occur with small but noticeable frequency. The reasoning behind the last criterion is that there is no point in detecting problems that never occur. Conversely, if a warning occurs too frequently, it is likely that it is not really causing any problems. In order to guarantee the usability of the tool, all interactions with analyzers are monitored, and if their usefulness measured against the aforementioned criteria drop below certain thresholds the analyzers are disabled and the analyzer writer is notified such that they are able to improve the performance. While the effective false positive rate is closely monitored, Tricorder, unlike Sensei, does not apply these restrictions on a project basis, but instead all analyzers are used company-wide. In Section 6.1 we compared the EFP rates for Tricorder and Sensei. Whereas only a 20% "Apply fix" rate is reported for Tricorder [15], with Sensei 98.2% of code markings were resolved by the developer. Of these 75.3% were fixed using the provided quick fixes in the IDE, and only 1.8% of code markings were ignored by Sensei users, which is a low EFP rate. After carefully improving their analyzers, Tricorder reached an EFP rate of around 5%. While both the customized rules of Sensei and the customized analyzers used by Tricorder appear to be effective solutions for preventing EFPs, quickfixes are more practical in the IDE than during the review stage, as is evident from the low "Apply fix" rate for Tricorder.

7.1.3. SpotBugs SpotBugs [34] (formerly FindBugs [35]) is a light-weight open-source analysis tool capable of finding a wide range of software bugs, including a number of security bugs [36]. SpotBugs' bug descriptions are very short, do not suggest any remediation but provide links to relevant Wikipedia articles. This lack of information and remediation to the developer has shown to result in low developer trust, as was explained in Section 6.1. Over half of the reported issues are never even reviewed.

SpotBugs allows the creation of third party "detectors", so additional security bugs can be included. These have to be implemented through an API [37] and compiled into a SpotBugs plugin. FindSecBugs [38] is a popular security plugin for SpotBugs. While the creation of additional detectors is not as convenient as with Sensei, the distribution through plugins does seem to be an effective way to encourage their creation.

SpotBugs is well researched [24, 7, 39] and used in industry. Despite its IDE integration, it is mostly used to scan after development due to their long scan times, which can take up to 20 minutes. In practice the tool is integrated in later parts of the SDLC and scan results of the code are often available a day or two after it has been committed to the version control repository [7].

SpotBugs provides support for so-called filter files. These files allow to manage the rules. They are used to configure the bug reports, including or excluding bugs on particular classes and methods.

Research by Ayewah et al. [24] showed that the tool has an EFP rate of 77%, and that the most interesting bugs were found and fixed without SpotBugs, namely after they were revealed by static analysis scans later in the SDLC. Ayewah et al. conclude, however, that the tool could have been used to discover those bugs earlier, if only it would have been used more actively by developers. This is in line with our earlier observation that low EFP rate inhibits effectiveness. We believe that shorter scan times, better descriptions, and remediation help as available in Sensei might improve the use of SpotBugs by developers in earlier stages of development.

7.1.4. SecureAssist SecureAssist [40] is an IDE plugin targeting the discovery of security bugs in code. It is available for eclipse, intelliJ, VisualStudio, RAD, and Spring Tool Suite [41]. Its scans are not in the IDE but on the enterprise portal. The results are sent back to the IDE once completed. This allows scanning without preventing the developer from continuing his work, which contrasts with the Fortify Security Assistant, that prevents developers from changing the code during a scan. Remediation is provided in the form of descriptions that explain the attack and provide some code examples but the tool does not provide quick fixes [42].

Rule packs can be added and created in their rule pack configurator. Rules are written in XML format instead of YAML but the syntax is user friendly and easily readable [43]. In Listing 10 an example rule for the SecureAssist plugin is shown to discover uses of the insecure DES cryptographic algorithm, as a comparison the same rule for the Sensei plugin is shown in Listing 11. SecureAssist's rule syntax is most similar to Sensei rules. However, creating the rules requires learning their exact syntax, as no editor is provided. Sensei on the other hand provides a rule wizard, context-aware suggestions, and a GUI to edit the rules. Sensei rules also support more comprehensive features such as the concept of untrusted variables and support for libraries, which were explained in detail in Section 4.

Rule packs are distributed as JAR files and the tool provides a Rulepack Configurator very similar to Sensei's Rule Manager.

7.1.5. Veracode Veracode offers two tools. The Veracode Static Analysis (VSA) is a SaaS platform that is similar to Fortify on Demand [44]. Veracode Greenlight (VG) is an IDE plugin, similar to Fortify Security Asisstant.

VSA performs static analysis scans on compiled bytecode of web applications in 23 programming languages. Because it does not need access to the source code it can also analyse frameworks and libraries used in the project. VSA provides integrations with popular ticketing systems, CI/CD tools and three IDEs: IntelliJ, Visual Studio, and Eclipse. All these integrations offer the possibility to start scans and download the results. Veracode focuses heavily on not only detecting vulnerabilities but also guiding remediation. To that extent, they provide detailed instructions and videos. There is even

```
<Match>
 <QualifiedName>javax.crypto.Cipher</QualifiedName>
 <Method>getInstance</Method>
 <Arguments>
  <Argument>
   <Index>0<Index>
   <Value>
    <ComparatorOperator>equals</ComparatorOperator>
    <ExpectedValue>DES</ExpectedValue>
    <ComparatorType>String</ComparatorType>
   </Value>
  <Argument>
 </Arguments>
</Match>
```

Listing 10: SecureAssist rule to discover use of DES

```
search:
 methodcall:
  name: "getInstance"
  type: "javax.crypto.Cipher"
  args:
   1:
    type: "java.lang.String"
    value: "DES"
```

Listing 11: Sensei rule to discover use of DES

the possibility to schedule a one-on-one conference call with a consultation expert. This expert can help the developer determine whether an issue is a false positive or what the best remediation is for a detected vulnerability. Scheduling such a consultation will usually take about three days. Veracode does not encourage or provide help to share the knowledge that is gathered by the developer during such a consultation session with their peers.

The company claims most scans finish in under an hour. This means the feedback cycle is rather long compared to the other tools. Since the scans are performed on binaries, they are not able to provide quickfixes as Sensei does, which is unfortunate for a solution otherwise very focused on remediation.

Veracode promotes a low false positive rate of 5% and hence discourages rule customization. Rule customization would also be much harder, since VSA analyzes compiled bytecode, not source code. VSA does not allow to disable rules or issues individually, but lets companies define a custom security policy for their application. Setting a policy configures the conditions used to evaluate results of a scan, and helps decide if the build should break or be accepted. In these policies it is possible to add rules that disable certain CWE entries, category of flaws, or severity of flaws. Veracode offers Program Management Support, where a Program Manager will help define these policies, and gradually increase their requirements. This helps guarantee developer usability during roll-out. Developers will first be encouraged to resolve the most severe issues. In later stages when a tighter policy is set, they are exposed to additional issues. Although a new policy does not provide as granular control as a new rule in Sensei, this feature and roll-out procedure closely resemble the best practices described in Section 6.2.

VSA offers a dashboard to track scan results over time. This displays the number of flaws per scan, tracks which ones are new, and organises the existing flaws into severity categories.

VG is an IDE plugin for Eclipse, IntelliJ, and Visual Studio. It is a light-weight version of VSA that is scanned locally. Since the scans are performed on bytecode, it requires a successful build. VG can be configured to scan automatically by using the Automatic Build option in the IDE. Scans are not real-time like Sensei but take around 3 seconds, after which the results are not shown in the IDE editor, but in a separate pane. Double clicking an issue in this pane will open the editor at the relevant location in code. Like VSA, VG is unable to provide quickfixes, but provides a lot of detailed information to the developer.

7.1.6. Checkmarx Checkmarx Static Application Security Testing (CxSAST) [45] is a static analysis tool that perfoms source code scans. It has support for over 25 coding and scripting languages, including Java, C#, and python. Similar to Fortify and Veracode, CxSAST has IDE plugins for Eclipse, Visual Studio, and IntelliJ. Again, and still in contrast to Sensei, these plugins do not perform any local scans but instead allow uploading the source code to CxSAST. They provide an interactive way to view the scan results by marking the relevant code in the IDE editor.

Checkmarx claims flexible rules lead to higher accuracy, and have a very extensive Query Language (CxQL) [46] to create and adapt rules. Managing queries is done in the CxAudit tool, this tool allows enabling and disabling individual queries. Here, new queries can also be added in a Query Source Pane. It is possible to export a set of queries from this tool to use in different applications. The source pane however provides little help to write and especially debug new rules. Since the rule-writing tool is independent from the scanning tool and the IDE, it requires long iterations to optimize rules compared to the instant feedback in the Sensei rule editor where the results of rules are updated live. Vulnerabilities marked in the scan results have a category but no descriptions are provided. This means that little help is provided compared to the remediation suggestions and quickfixes of Sensei. More details about the control flow leading to the vulnerability are clearly marked in the IDE plugins and can help an experienced developer find the best place in source code to resolve the problem. A similar feature could be developed for Sensei's concept of trusted input, but it was decide to omit this to avoid unnecessary clutter in the IDE.

Checkmarx does provide an integration between CxSAST and its training service CxCodebashing. This way developers can learn about the vulnerability and how to fix it in an interactive way. This integration is similar to the integration between Secure Code Warrior and Fortify on Demand, as well as the training links in the full coding guideline description of Sensei.

7.1.7. Snyk Snyk [47] is a tool designed to monitor and fix insecure dependencies. It will not look for vulnerabilities, but will mark dependencies with known vulnerabilities and helps to remediate by updating or replacing dependencies. It started as a plugin for code repositories, such as github, gitlab, and bitbucket. Fixes in these plugins are in the form of automatic pull requests that fix the dependencies. Following the shift left movement Snyk also developed IDE plugins for intelliJ and VS Code to detect insecure dependencies earlier in the SDLC. The tool is suitable for use in the development phase since scanning for insecure dependencies is light-weight and it is mostly trivial to propose quickfixes.

7.1.8. Aside The OWASP ASIDE/ESIDE [48] project consist of two branches, the ASIDE branch that focuses on detecting software vulnerabilities and helping developer write secure code, and the ESIDE branch that focuses on helping students in acquiring secure programming knowledge and practices.

Application Security IDE (ASIDE) performs fast scans of the code in Eclipse, but unlike Sensei the scans need to be started manually. Besides detecting vulnerabilities they also provide quick fixes for some issues. The quick fixes require the developers to choose from a list of options, which could overwhelm them. In previous research a large number of false positives were detected [49], however, most of these are what is considered protection for future use in this paper. They are cases where best practices should be applied even if their violation is not yet exploitable at this point in development. They also mark variables in the code that are tainted, this could be considered to Sensei's concept of untrusted input. Untrusted input in Sensei is not currently marked to avoid unnecessary clutter.

The goal of ESIDE [50, 13] is to provide information and training at all times during the education. Its rules can not be configured and the tool does not provide quickfixes. However they provide explanation in external web pages linked from Eclipse. Their information is similar to our full coding guidelines where information on APIs and a correct code example is provided.

7.2. *Penetration Testing*

Penetration testing is the practice of breaking into running software by attacking it. Sometimes the penetration tester has access to the source code to speed up this process. It is a common practice used by many companies and usually external experts are hired to perform these tests [51]. Since the penetration tester needs access to the running software this can only be done very late in the SDLC. Already in the introduction, we addressed that relying on security experts does not scale well. Furthermore, it does not integrate well in agile or DevOps development strategies [52]. Penetration testing improves the security awareness of the developers but does not cause any long-lasting change of development practices by itself [53].

7.3. *Static Analysis*

As discussed in Section 2.1 static analysis tools are typically used in a reactive approach to security. A shift left movement is ongoing to apply these tools as early as possible in the SDLC. Static analysis tools are well researched [54, 55, 56] and commonly used to detect vulnerabilities [51]. Most tools can run automatically and as a result are easily adapted by agile and DevOps teams. Static analysis tools vary from robust and time-consuming analyses such as Fortify [57] to light real-time analyses [58]. In controlled experiments, static analysis tools proved to be more effective than penetration testing [59]. However, there are some downsides to their global analyses that can be mitigated with our approach. Static analysis tools can not be trivially tailored by ignoring context. For example, vulnerability testing with context-insensitive, local versions of the analyses will in many cases not be as effective in terms of EFP rates as rules that are designed and tuned for local analyses. And even if local versions of the analyses perform well, they do not provide fixes to resolve the discovered issues and hence do not enforce uniform solutions like Sensei does.

Some techniques have been developed to improve the speed of static analysis such that they require orders of magnitude less code to perform similar analyses [58]. One such technique is the use of micro grammars as checkers. These are incomplete, language-independent parsers that can be used to describe patterns in code. This technique is vastly less complex than traditional static analysis. The focus is on language errors (e.g., unchecked dereferences), but has never been applied to security bugs. Security bugs are more language-specific and API-dependent than language errors so this approach can not be easily applied to security. However, we have considered making Sensei rules virtual-machine dependent instead of language dependent. For example Java, Kotlin, and Scala all run on the Java Virtual Machine (JVM). They make use of the same APIs but use different syntax. It is possible to imagine JVM rules that are applicable to all three languages.

7.4. *Code Reviews*

Code review is a manual inspection of produced code. It is usually done by another developer than the original author but with that author present. The advantage of this practice is that it can be done as early as desired in the SDLC. Code reviews also provide an educational aspect for the developer whose code is reviewed [60]. The downside is that, similarly to penetration testing, it relies on internal or external experts and hence does not scale well.

Some tools are designed to automate the detection of issues in the review stage, such as Tricorder [15] and Snyk (https://snyk.io/). While these can provide fixes, most developers go back to their IDE rather than use the code review tool to resolve the issues [15].

7.5. *Security Patterns*

Adherence to coding guidelines leads to secure code [61]. For this reason there is plenty of research into secure patterns. Some of the resulting guidelines are generic and high level [62]. In order to support these with our plugin, they need to be translated into concrete guidelines and customized for the used APIs.

Others provide clear API-level instructions that can directly be implemented as rules in our plugin. We have demonstrated this by creating a rule set from The Android Application Secure Design/Secure Coding Guidebook by the Japan Smartphone Security Association [63].

Another notable example is the guidelines designed to counter side-channel attacks, designed by Witteman [64]. The Java code issues and transformations they discuss fall clearly within the capabilities of our plugin.

Finally, some efforts have also been made to automatically generate rules from code changes such as Paletov et al. [25]. This is something we intend to research to improve and speed up roll out of Sensei in companies and projects.

7.6. *Security Libraries and Frameworks*

Another solution to make developers adhere to coding guidelines, is by implementing them into frameworks or libraries. An example is the OWASP ESAPI. It is an open source application security control library that provides some very clear replacement APIs for insecure JDK implementations [19]. As mentioned in the paper, our plugin and rule sets already provide support for replacing banned methods from the ESAPI. Other parts of the ESAPI still need to be implemented. We foresee no challenges in covering them with the plugin.

Recently, the most popular web application frameworks started automatically sanitizing inputs to prevent common vulnerabilities. This has shown to be effective at preventing security problems as indicated by the position of XSS in the OWASP top 10. XSS has moved to seventh place in 2017 from third place in 2013, even though XSS attempts remain common [8]. We have observed that these efforts result in useful code examples that ease the development of rules and quick fixes for coding guidelines regarding input validation and output escaping.

7.7. *Lint*

Linter tools are designed to allow the developer to concentrate at solely on the algorithms, data structures, and correctness of the program, and only later, with the aid of lint, address non-functional aspects of the code. They mostly focus on syntax and styleguide checking but some tools are advanced enough to check for certain bugs as well. Depending on their targets linters perform their analyses with string-matching or reduced versions ASTs without symbol information. The more advanced lint tools perform similar analyses to Sensei, making use of the entire AST. Some examples of lint tools supporting Java include Error Prone (http://errorprone.info/), checkstyle (http://checkstyle.sourceforge.net/), PMD (https://pmd.github.io/), and SonarLint (https://www.sonarlint.org/). Of the named lint tools only two have rules for security, PMD has two security rules and SonarSource has 48. Many lint tools are open-source which means their rules can be customized, however, none are designed for easy and fast customization of the rules.

8. CONCLUSIONS AND FUTURE WORK

We stated the requirements to build a successful tool to enforce secure coding guidelines. Enforcing coding guidelines has some clear advantages compared to conventional static analysis solutions because it can be done without the need for context. The approach is more proactive and can be applied earlier in the SDLC. This also makes it more feasible to hand the developer solutions in the form of quick fixes. Since the context is not needed and all checks can be done using local analyses which avoids performance issues encountered by other tools. Easy customization of rules greatly improves knowledge distribution in the team and allows guidelines to be tailored to specific projects which improves the usability. We also discussed several ways to keep the developer's trust in the tool by lowering the EFP rate and making sure the right information is presented.

To improve the plugin we want to research how to increase the up-take of the generators. Research directions of interest include making them more context aware. Next, rule writers should be allowed to create tests for their rules, preferably this process will be mostly automated. Finally, we want to apply existing patch generation techniques in order to automatically generate new Sensei rules.

In future work we also plan to evaluate the tool more thoroughly by setting up additional empirical experiments. Our goal is to prove its effectiveness at preventing security problems, validating its usability and applicability to different projects.

Finally we want to verify whether this approach can be adapted to lower-level languages like C and C++ and evaluate to what degree the learned lessons are still applicable. Support for such languages is more challenging, among other because they are pre-processed and because they feature pointers.

ACKNOWLEDGEMENT

The research by Pieter De Cremer at Ghent University and Secure Code Warrior is funded through a Baekeland mandate PhD grant from the Flanders Innovation and Entrepreneurship Agency (VLAIO) with project number HBC.2017.0215.

REFERENCES

1. US Department of Homeland Security. Infosheet Software Assurance. `https://www.us-cert.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf`. Last accessed 2018-05-22.
2. Xie J, Chu B, Lipford HR. Idea: Interactive support for secure software development. ESSoS, Springer, 2011; 248–255.
3. McGraw G, Migues S, West J. Building security in maturity model (bsimm). `https://www.bsimm.com/` 2018.
4. Damm LO, Lundberg L, Wohlin C. Faults-slip-through—a concept for measuring the efficiency of the test process. Software Process: Improvement and Practice 2006; **11**(1):47–59.
5. Briand LC, El Emam K, Freimut BG, Laitenberger O. A comprehensive evaluation of capture-recapture models for estimating software defect content. IEEE Transactions on Software Engineering 2000; **26**(6):518–540.
6. Baca D, Carlsson B, Lundberg L. Evaluating the cost reduction of static code analysis for software security. Proc. 3rd ACM SIGPLAN workshop on Programming languages and analysis for security, ACM, 2008; 79–88.
7. Ayewah N, Pugh W. The google findbugs fixit. Proc. 19th Int'l Symp. on Software testing and analysis, 2010; 241–252.
8. Trustwave. Global Security Report. `https://www2.trustwave.com/rs/815-RFM-693/images/Trustwave_2018-GSR_20180329_Interactive.pdf`. Last accessed 2018-05-22.
9. Xie J, Lipford HR, Chu B. Why do programmers make security errors? Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on, IEEE, 2011; 161–164.
10. Sharma A, Misra PK. Aspects of enhancing security in software development life cycle. Advances in Computational Sciences and Technology 2017; **10**(2):203–210.
11. Layman L, Williams L, Amant RS. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, IEEE, 2007; 176–185.
12. Syed M. Black Box Thinking: Why Most People Never Learn from Their Mistakes–But Some Do. Penguin, 2015.
13. Whitney M, Lipford HR, Chu B, Thomas T. Embedding secure coding instruction into the ide: Complementing early and intermediate cs courses with eside. Journal of Educational Computing Research 2018; **56**(3):415–438.
14. Banerjee C, Pandey S. Software security rules, sdlc perspective. arXiv preprint arXiv:0911.0494 2009; .
15. Sadowski C, Van Gogh J, Jaspan C, Söderberg E, Winter C. Tricorder: Building a program analysis ecosystem. Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, 2015; 598–608.
16. Baset AZ, Denning T. Ide plugins for detecting input-validation vulnerabilities. 2017 IEEE Security and Privacy Workshops (SPW), IEEE, 2017; 143–146.
17. Xie J, Lipford H, Chu BT. Evaluating interactive support for secure programming. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2012; 2707–2716.
18. Johnson B, Song Y, Murphy-Hill E, Bowdidge R. Why don't software developers use static analysis tools to find bugs? Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013; 672–681.
19. OWASP Enterprise Security API Toolkits - Datasheet. `https://www.owasp.org/images/8/81/Esapi-datasheet.pdf`.
20. Bessey A, Block K, Chelf B, Chou A, Fulton B, Hallem S, Henri-Gros C, Kamsky A, McPeak S, Engler D. A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM 2010; **53**(2):66–75.
21. Romeo C. How to Transform Developers into Security People. `https://www.rsaconference.com/videos/how-to-transform-developers-into-security-people`.
22. Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y. Evaluating static analysis defect warnings on production software. Proc. 7th workshop on Program analysis for software tools and engineering, 2007; 1–8.
23. Tabassum M, Watson S, Lipford HR. Comparing educational approaches to secure programming: Tool vs. ta. Symposium on Usable Privacy and Security (SOUPS), 2017.
24. Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y. Using findbugs on production software. Companion to the 22nd Conf. on Object-oriented programming systems and applications, 2007; 805–806.
25. Paletov R, Tsankov P, Raychev V, Vechev M. Inferring crypto api rules from code changes. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2018; 450–464.
26. Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming. Proc. 31st Int'l Conf. on Software Engineering, IEEE Computer Society, 2009; 364–374.
27. Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013; 802–811.

28. Long F, Rinard M. Automatic patch generation by learning correct code. ACM SIGPLAN Notices 2016; **51**(1):298–312.
29. Micro Focus. Fortify Static Code Analyzer: Static Application Security Testing. `https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview`. Last accessed 2019-10-14.
30. Micro Focus. Fortify SCA Custom Rules Reference. `http://bigsec.net/b52/Fortify/rules-schema/`. Last accessed 2019-10-15.
31. Micro Focus. Fortify on Demand: Application Security as a Service. `https://www.microfocus.com/en-us/products/application-security-testing/overview`. Last accessed 2019-10-15.
32. Micro Focus. Secure SDLC - IDEs. `https://www.microfocus.com/en-us/marketing/secure-sdlc-and-devops#section3`. Last accessed 2019-10-15.
33. Google. Github: Shipshape. `https://github.com/google/shipshape`. Last accessed 2019-10-15.
34. SpotBugs. SpotBugs: Find bugs in Java Programs. `https://spotbugs.github.io/`. Last accessed 2019-10-15.
35. University of Maryland. FindBugs™ - Find Bugs in Java Programs. `http://findbugs.sourceforge.net/`. Last accessed 2019-10-15.
36. SpotBugs. Bug descriptions. `https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#security-security`. Last accessed 2019-10-15.
37. SpotBugs. SpotBugs API Documentation. `https://javadoc.io/doc/com.github.spotbugs/spotbugs/3.1.10`. Last accessed 2019-10-15.
38. FindSecBugs. Find Security Bugs: The SpotBugs plugin for security audits of Java web applications. `https://find-sec-bugs.github.io/`. Last accessed 2019-10-15.
39. Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W. Using static analysis to find bugs. IEEE Software Sept 2008; **25**(5):22–29, doi:10.1109/MS.2008.130.
40. synopsys. SecureAssist Overview. `https://community.synopsys.com/s/article/SecureAssist-Overview`. Last accessed 2019-10-25.
41. synopsys. SAST in IDE (SecureAssist). `https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/secureassist-datasheet.pdf`. Last accessed 2019-10-25.
42. synopsys. How to use SecureAssist IntelliJ Plugin. `https://community.synopsys.com/s/article/How-to-Use-SecureAssist-IntelliJ-Plug-in`. Last accessed 2019-10-25.
43. synopsys. SecureAssist Custom Rule Tutorial. `http://download.asteriskresearch.com/2.4/SecureAssist%20Custom%20Rule%20Tutorial%2010-2014.pdf`. Last accessed 2019-10-25.
44. Veracode. Veracode Static Analysis: Don't Just Find Security Defects in Your Code - Fix Them Fast. `https://www.veracode.com/products/binary-static-analysis-sast`. Last accessed 2019-10-15.
45. Checkmarx. Static Application Security Testing: Secure Your Code from the Very Beginning. `https://www.checkmarx.com/products/static-application-security-testing/`. Last accessed 2019-10-15.
46. Checkmarx. CxAudit Overview. `https://checkmarx.atlassian.net/wiki/spaces/KC/pages/5406733/CxAudit+Overview`. Last accessed 2019-10-15.
47. Snyk. Open Source Security Platform. `https://snyk.io/`. Last accessed 2019-10-22.
48. OWASP. ASIDE Project. `https://www.owasp.org/index.php/OWASP_ASIDE_Project`. Last accessed 2019-10-16.
49. Xie J, Chu B, Lipford HR, Melton JT. ASIDE: IDE support for web application security. Proceedings of the 27th Annual Computer Security Applications Conference, ACM, 2011; 267–276.
50. UNC Charlotte College of Computing and Informatics. Educational Security in the IDE (ESIDE). `https://eside.uncc.edu/`. Last accessed 2019-10-16.
51. Cruzes DS, Felderer M, Oyetoyan TD, Gander M, Pekaric I. How is security testing done in agile teams? a cross-case analysis of four software teams. Int'l Conf. on Agile Software Development, Springer, 2017; 201–216.
52. Cruzes DS, Felderer M, Oyetoyan TD, Gander M, Pekaric I. How is security testing done in agile teams? a cross-case analysis of four software teams. Agile Processes in Software Engineering and Extreme Programming, Baumeister H, Lichter H, Riebisch M (eds.), Springer International Publishing: Cham, 2017; 201–216.
53. Türpe S, Kocksch L, Poller A. Penetration tests a turning point in security practices? organizational challenges and implications in a software development team. WSIW@ SOUPS, 2016.
54. Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Traon L. Static analysis of android apps: A systematic literature review. Information and Software Technology 2017; **88**:67–95.
55. Jovanovic N, Kruegel C, Kirda E. Pixy: A static analysis tool for detecting web application vulnerabilities. Security and Privacy, 2006 IEEE Symposium on, IEEE, 2006; 6–pp.
56. Livshits VB, Lam MS. Finding security vulnerabilities in java applications with static analysis. USENIX Security Symposium, vol. 14, 2005; 18–18.
57. Chess B, McGraw G. Static analysis for security. IEEE Security & Privacy 2004; **2**(6):76–79.
58. Brown F, Nötzli A, Engler D. How to build static checking systems using orders of magnitude less code. ACM SIGPLAN Notices 2016; **51**(4):143–157.
59. Scandariato R, Walden J, Joosen W. Static analysis versus penetration testing: A controlled experiment. 24th Int'l Symp. on Software Reliability Engineering (ISSRE), 2013; 451–460, doi:10.1109/ISSRE.2013.6698898.
60. Futcher L, von Solms R. Guidelines for secure software development. Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, ACM, 2008; 56–65.
61. Lipford HR, Xie J, Stranathan W, Oakley D, Chu BT. The impact of a structured application development framework on web application security. Proceedings of the 14th Colloquium for Information Systems Security Education, Baltimore Marriott Inner Harbor, 2010; 212–219.
62. Schumacher M, Fernandez-Buglioni E, Hybertson D, Buschmann F, Sommerlad P. Security Patterns: Integrating security and systems engineering. John Wiley & Sons, 2013.
63. Association JSS. Android Application Secure Design/Secure Coding Guidebook. `http://www.jssec.org/dl/android_securecoding_en.pdf` 2016. Last accessed 2018-05-22.

64. Witteman M, Oostdijk M. Secure application programming in the presence of side channel attacks. RSA conference, vol. 2008, 2008.
65. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. Experimentation in software engineering. Springer Science & Business Media, 2012.

A. RULE SCOPES

The **class scope** can enable or disable rules based on the name and/or package of the class itself or based on the name and package of any classes or interfaces it inherits from.

The **method scope** enables rules based on the name of the method. These scopes are mostly used to enforce guidelines in inherited methods. For example, when creating a servlet in Java EE, it is advised to configure some security headers with the `doGet` and `doPost` methods inherited from the `HttpServlet` class. We do this by enforcing a guideline that states that the `addHeader` method should be called with specific parameters to set the required headers. We then limit the scope of this guideline to only be enabled when the class inherits from `HttpServlet` and the method name is `doGet` or `doPost`. Using the YAML syntax many properties of the method can be used for the scope, such as the number of parameters, the types of parameters, the return type, and any annotations added to the method.

The **file scope** is used to enable rules based on project file names. This is mostly used for configuration files. This allows us for example to enforce coding guidelines in the Android manifest file, as its name is always `AndroidManifest.xml`. This scope is not yet migrated to the YAML syntax.

The **Android context scope** was created to raise context awareness in Android projects. In the Android manifest a developer can configure capabilities of components such as activities and broadcast receivers regarding their communication towards the OS. They can listen to any other application, or only to authorized applications, or only to the own application. The Android context scope allows us to enable rules based on the configuration of the relevant component, so that we can enforce different rules for different levels of exposure. We can for example allow communication of sensitive information between classes that are configured as private components, but not between other classes. This scope is not yet migrated to the YAML syntax.

The **Android build property scope** can be used to enable rules based on the build property of an Android project. Mostly this is used to look at the `minSdkVersion` property, to determine what versions of Android the application will be compiled to. Specific version of Android have specific vulnerabilities, so rules need to be disabled based on that build properties. This scope is not yet migrated to the YAML syntax.

B. EMPIRICAL USABILITY EXPERIMENT

In the evaluation in Section 6.1 we discussed the results of a controlled experiment where the usage statistics of 32 users were analyzed. The findings were then compared to those of other experiments in literature. In this appendix we describe the experiment in more detail, discussing its goals and parameters, the experimental procedure, results, and possible threats to validity.

B.1. Goals and Research Questions

The main *goal* of the experiment is to observe the impact of the Sensei IDE plugin on developers and developed code during development. The *purpose* is evaluating the usability and effectiveness of Sensei in supporting the security by design development strategy. The *quality focus* is the ability of the plugin to prevent developers from violating secure coding guidelines without causing a significant cognitive burden. The study evaluates the behaviour of a developer. We aim at measuring the increase in cognitive burden when developers use the plugin, by measuring the impact on development time. In our experiment we evaluated the impact on a group of students who have a consolidated minimum level of expertise in both web application development and web application security.

The above goal can be achieved by means of an experiment aimed at answering the following four questions:

- **RQ1** How effective are the Sensei code markings at grabbing the developer's attention?
- **RQ2** Does the plugin significantly impact the development time?
- **RQ3** Do developers often use the provided programming aid (quickfixes) to resolve code markings?
- **RQ4** Are there any specific code markings that significantly impact the usability compared to others?

B.2. Subjects

The subjects for this study are a group of third year students following the Bachelor program for Computer and Cyber Crime Professional (https://www.howest.be/en/programmes/bachelor/applied-informatics/computer-and-cyber-crime-professional) at the college Hogeschool West-Vlaanderen (Howest) in Bruges, Belgium. All students are in the third, and final year of the bachelor. The experiment was performed in the context of the course Secure Object Oriented Architectures. In this course the students are taught design patterns, how to design three layered applications, and java technicalities. During the entire course the focus is on development while conforming to Oracle's Secure Coding Guidelines (https://docs.oracle.com/cd/E26502_01/html/E29016/scode-1.html).

All of the students are familiar with Java programming in IntelliJ IDEA, as it is the main language and IDE used in the education program. They are, however, not experienced or trained with the Sensei tool. Instead this experiment was their first exposure to the tool.

The experiment was preceded by a secure coding tournament using the Secure Code Warrior platform. The goal of this tournament was to both engage the students as well as measure their skill level. Student participation was voluntary, out of the 75 students that participated in the tournament, 60 also participated in the experiment itself. However, only 32 students successfully submitted all necessary files after the experiment, see Section B.6.2.

B.3. Development task

The subjects were given a development task to complete with the Sensei plugin installed in the IDE. For the assignment they received the incomplete code for an employment web app. The application provides employees of a company a way to view, download, and upload their payslips, as well as to submit requests for absence. The application is written in Java and uses Java Server Pages (JSP) as the server side technology. Some of the features are incomplete and must be completed by the subjects during the experiment. During the implementation of these features the subjects are at risk of introducing a number of web application vulnerabilities. Below is a list of features to be completed and their associated risks.

- A web page to view absence requests: risk of cross site scripting (XSS).
- A web from to search for absence requests in the database: risk of SQL injection.
- A web form to upload payslips in XML format: risk of XML injection, XML external entity, unrestricted file upload, and local file inclusion.
- Log all attempts made on the sign in page: risk of log forgery.

B.4. Treatment

B.4.1. Tournament One week preceding the experiment all subjects participated in a tournament on the Secure Code Warrior platform. On this platform they are tasked to identify, locate, or fix vulnerabilities in code samples. The subjects were handed 24 secure coding exercises in Java using JSP to complete within 90 minutes. The maximum score to gain per successful exercise depends on the difficulty of the exercise:

- Easy: 100 points
- Medium: 200 points
- Hard: 300 points

For each exercise the maximum attempts is three. The score gained per successful exercise decreases with the amount of attempts:

- Correct on first attempt: 100% awarded
- Correct on second attempt: 60% awarded
- Correct on third attempt: 30% awarded

The subjects also had access to hints to help them solve the exercises, each hint used decreases the maximum score to be gained from that exercise:

- **Locate exercises**
  - Maximum score after 1 hint: 100%
  - Maximum score after 2 hints: 95%
  - Maximum score after 3 hints: 60%
  - Maximum score after 4 hints: 0%
- **Identify exercises**
  - Maximum score 1 hint: 100%
  - Maximum score 2 hints: 50%

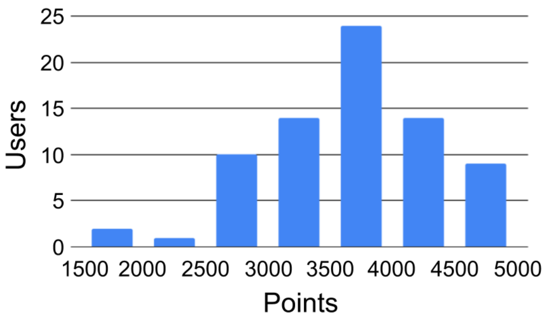


Figure 16. The histogram shows that the points scored by participants of the tournament is approximately a normal distribution around the mean of 3659 points.

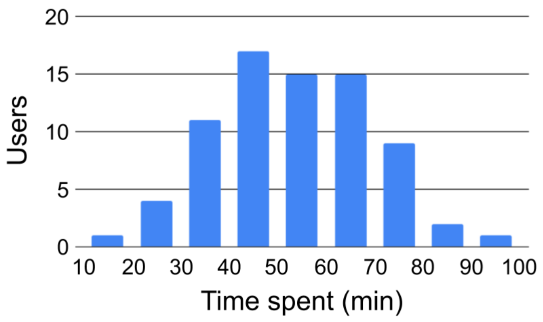


Figure 17. The histogram shows that the time spent by participants (n=75) competing in the tournament tournament is approximately a normal distribution around the mean of 53.70 min.

  - Maximum score 3 hints: 0%

- **Fix exercises**
  - Maximum score after 1 hint: 66%
  - Maximum score after 2 hints: 33%
  - Maximum score after 3 hints: 0%

The total maximum score for all exercises in the tournament is 5000 points. The highest score reached was 4920, while the lowest was 1600. The mean score reached by the participants (n = 75) was 3659 (s = 710). The mean time spent solving all exercises was 53.70 min (s = 16.07 min). Both the score and the time spent are approximately normal distributions as shown in Figure 16 and Figure 17. The score reached by each subject in this tournament was used to split the subjects into two equally skilled groups, a control group and a test group. The subjects were told that they were participating in an experiment regarding the Sensei plugin. They were told that they were split in a control group and test group but were not informed of which group they were part, or what the difference in treatment would be.

B.4.2. Sensei To complete the programming exercise the subjects on both groups were allowed to use their own device and operating system but they had to develop using the IntelliJ IDEA with the Sensei plugin installed (such that the control group was not aware of the differences in treatment). The Sensei installation of both the control group and the test group included a set of carefully tailored rules to prevent introduction of the vulnerabilities described in Section B.3. However, for the control group the markings and programming aid were disabled and the plugin was only used as a monitoring tool. All features to view, edit or disable rules were hidden, so that none of the subjects were able to alter the rules. The information available to the subjects, in the different descriptions, was designed as outlined in Section 3.4. In fact the example given in Figure 12 is a guideline used during the experiment.

B.5. Ethical Review Board

The teaching staff proposed the experiment to the college's ethical review board. We helped them in writing a detailed explanation of the activities and the goals of the experiment. The board approved the experiment under two conditions. Firstly, experiment participation was to be voluntary and students were not to receive extra credit upon participation. Secondly, all data handed to the researchers was to be made completely anonymous. We and the teaching staff then operated in line with these conditions.

B.6. Experimental Procedure

B.6.1. Controlled Experiment All subjects were allowed to user their own devices, and any resources they would normally use during development, such as books and internet access. We did not allow communication with other subjects. Students were allowed to take breaks and leave the room. However, as to not give incentive to finish hastily and without care, all students were required to be present during debriefing.

The subjects were given:

- a consent form to acknowledge that their data will be analysed anonymously;
- a repository URL to install the Sensei plugin, which automatically includes the set of rules for each subject;
- a link to an archive containing the IDE project for the assignment;
- plugin installation instructions;
- a detailed description of the programming assignment.

During the controlled experiment, we asked the subjects to complete the assignment using the procedure below:

1. open the plugins menu in the IDE and copy-paste the url to the plugin repository
2. install the plugin and restart the IDE after the process has completed
3. verify correct installation of the plugin by finding the "Sensei" menu in the menu bar of the IDE
4. download the archive containing IDE project
5. extract the archive and open it with the IDE
6. execute the project and read the messages in the console
7. open a web browser and browse to `localhost` to verify that the project is running correctly
8. sign in using provided credentials and get familiar with the functionality of the web application
9. read the description of the features to be implemented
10. complete the assignment in silence

Throughout all phases of the experiment, we provided assistance to the subjects and answered all questions unrelated to the security of their code or the information displayed by the plugin. Indeed, despite testing on several operating systems and IDE versions there were some setup issues to solve.

B.6.2. Post-Experiment Information Gathering When the task had been completed or the allocated time had ended, the subjects were instructed to:

1. navigate to the Sensei installation folder and find the Sensei events file, which contains a log of all the action monitored by the Sensei plugin
2. archive both the events file and the source files into one archive
3. submit the archive to the teaching staff

The events file contains timestamps and guidelines for all logged events. An example events file is shown in Table I. The events in the table include four newly introduced guideline violations (ADD). All four of the newly introduced guideline violations are removed (DELETE), two of them by using the quickfix (FIX). For some but not all guidelines we can also log code that is compliant (C_ADD), this highly depends on how strict the guideline is and if we can create an additional Sensei rule to detect all possible compliant code fragments. Finally, the events include the opening of one description (DESCRIPTION). The events file does not include code or code locations, as our clients do not want to expose this information.

Several days after the experiment, all data was handed to us by the teaching staff after having obscured all personal data. At this point we discovered that the hand-in procedure was not correctly performed by all subjects, as the majority of the subjects had handed in either the code or the events file but few handed in both as requested. We asked all subjects to hand in again, stressing to include both the events file and all source files, but very few subjects submitted a second time.

Without the source code in addition to the events file we are unable to verify whether the code is still functional, as simply removing the relevant pieces of code would also effectively remove all guideline violations. On the other hand, without the events file we can not verify which impact of the Sensei plugin has on the security of resulting code.

Event type	Timestamp	Rule ID
ADD	1540555403436	requestparam
DELETE	1540555463210	requestparam
C_ADD	1540555463219	requestparam
ADD	1540555587165	requestparam
DESCRIPTION	1540555599165	requestparam
DELETE	1540555615181	requestparam
C_ADD	1540555615186	requestparam
ADD	1540558842105	xss
ADD	1540558843108	xss
FIX	1540558843374	xss
DELETE	1540558843700	xss
FIX	1540558852374	xss
DELETE	1540558852890	xss
C_ADD	1540569018675	sqlinjection

Table I. Example of the events logged by Sensei with one existing guideline violation in legacy code and the introduction and removal of four new violations.

As a result we do not have sufficient data to compare result from both groups to evaluate the *effectiveness* of Sensei on improving the security of the final code. It remains future work to rerun a similar experiment including the proper collection of all required data. With the available data, however, we can still evaluate the *usability* of Sensei for the 32 subjects that have at least submitted their events file.

B.7. Analysis Method

Since the logs in the events file do not include file locations we sometimes have to make assumptions on which ADD and DELETE events should be paired. Sometimes there are multiple guideline violations with the same rule ID present in the code at a certain time, as is the case in Table I for the XSS rule ID. In this case we can not know for certain which of the two violations is fixed first. During our experiment, this was the case for 8% guideline violations, with the two ADD events on average 37.75 s (s = 44.05 s) apart. For the measurements of the time between adding the violation and removing it, we assumed that the violations were removed in the same order as they were introduced. For all of the cases eventually either both violations were removed or neither of them were. That means that the aforementioned assumption has no influence on the mean removal time and only influences the standard deviation of the removal time.

We observed three exceptionally long removal times and inspected the logs to determine the cause of this duration. Two of the outliers had events regarding other rule IDs in between the ADD and DELETE events and so the developer did not spend this time actively solving the guideline violation. For further computations of removal time, these two outliers are left out. In between the ADD and DELETE events of the third case there were a number of DESCRIPTION events with the same rule ID, in this case we can safely assume that the developer did indeed spend 3.54 min actively resolving the issue.

B.8. Results

B.8.1. Guideline violations On average, the subjects introduced 17.64 (s = 10.27) guideline violations. The best performing subject (in this regard) introduced 2 guideline violations. For this subject the events log showed enough C_ADD events to assume that the subject completed at least the majority of the programming exercise. The worst performing subject added added 37 guideline violations. In this case the events log showed a large number of ADD and DELETE events for the same rule ID, making us believe that the subject was rewriting the code a number of times. This can result from attempting to implement the code functionally correct or from attempting to resolve the guideline violation. The absence of DESCRIPTION events in the log is strong evidence for the former.

B.8.2. Resolving guideline violations Out of all the coding guideline violations 98.4% have been removed eventually. Out of the removed violations, 73.3% have been removed with a quickfix. For the remaining removals, we don't know the intention of the subject, i.e., whether the violations were resolved manually as the subject spotted them as violations or whether the removal was part of rewriting (or removing) the code for another reason, such as simply meeting the functional requirements of the assignment. The four unresolved guideline violations each violated one different guideline, so there was no particular guideline

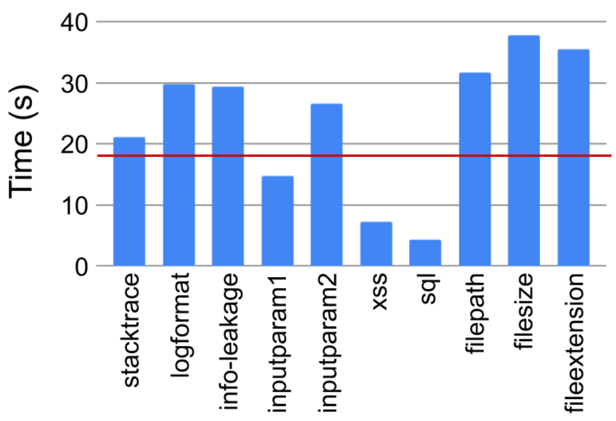


Figure 18. The graph shows that the average removal time for each guideline fluctuates heavily and many are different from the overall mean removal time of 19.10 seconds marked with a horizontal red line.

causing the majority of usability problems. One was violating a SQL query guideline and the others were violations of several file upload guidelines by the same user.

Out of the resolved violations, 89.3% were resolved within one minute, and 99.5% were resolved within three minutes. Only one case, previously discussed in Section B.7, took 3.54 min to resolve. This subject did eventually not use the quickfix to resolve the issue. On average the subjects took 19.10 s (s = 25.22 s) to resolve an issue. This large standard deviation is explained by a large difference in removal time for certain guidelines, as can be seen in Figure 18. On average commonly known vulnerabilities such as SQL injection and XSS are resolved within less than 10 seconds, while the guidelines regarding file upload vulnerabilities take significantly longer. Besides the familiarity with he vulnerability this difference can also explained by the fact that one piece of code can violate multiple guidelines. This was often the case for the file upload guidelines, the naive implementation without any security checks violates guidelines regarding file path, file size, and file extension. The developers violating these guidelines receive a lot of simultaneous feedback, which takes longer to process.

All of the subjects used at least one quickfix, with an average of 12.71 (s = 4.73) quickfixes used per subject. Less than half of the users (42.85%) have opened a description. On average the subjects opened 2.79 (s = 7.64) descriptions.

B.8.3. Development Time Using the events file from Sensei we can determine the approximate development time for the entire experiment. If we take the time difference between the first and the last event this will likely be close to the total development time. This can be done for all users of both the control and the test group that handed in the events file. For users in the test group we can also approximate the time spent addressing Sensei markings by taking the sum of all removal times. We can compare these results to see how much impact Sensei has on the development time. Users of the control group (n = 17) spent on average 61.54 min (s = 17.68 min) to complete the experiment. Users of the test group (n = 15) spent on average 68.75 min (s = 14.42 min). This is an increase of 11.72% in development time when using the plugin. The time spent by the test group addressing coding guideline violations was on average 8.42 min (s = 10.06 min). The average share of total development time that is spent addressing guideline violations is 11.28% (s = 12.25%). This is consistent with the previously measured 11.72% increase in development time.

B.9. Threats to Validity

We check our experiment against the possible threats to validity as proposed by Wohlin et al. [65].

B.9.1. Conclusion Validity The final score of each subject in the tournament is not a complete estimate of the subject's skills regarding security or secure development. Since there is a time limit, a good score is also partly achieved by time management. One one hand, taking too much time to complete the exercises will result in missed scoring opportunities by not finishing all exercises. On the other hand, answering too hastily may result in mistakes that otherwise could have been avoided, again resulting in a loss of points. However, the exercises were in the same language and framework as the development task, and the subjects also had a limited time to complete this task, so it is a reasonable estimate.

Each group of subjects were given the exact same development exercise, only different treatment.

The subjects were not heterogeneous, as they were all bachelor students, and the tournament score was used to avoid random irrelevance to some degree.

B.9.2. Internal Validity Before starting the experiment, we clearly explained the programming assignment and answered any arising questions publicly. The experiment itself was conducted in a single session, with all participants in the same room, this excludes all threats related to location, and repetitions.

Since the experiment was preceded by a secure coding tournament, and the experiment takes place in a security oriented class, this history can affect the experimental results. However, do note that the entire bachelor's program followed by the subjects is focused on security, so the security related activities are not that different from usual day-to-day activities.

Since the experiment took over an hour, depending on the speed of development, subjects may react differently as time passes. Indeed, to avoid students getting tired, bored, or frustrated we allowed them to take breaks and leave the room. We also note that the opposite is possible, and even likely, the subjects could have been learning and adjusting their behaviour during the experiment. This will also interact with the selection, since the test group receives feedback on their behaviour through the tool, and the control group does not.

The effect of letting volunteers take part in an experiment may influence the result, since they are generally more motivated and suited for a new task than the whole population. The subjects group might not be representative for the whole population.

Since some of the subjects did not hand in their Sensei events file, it can be useful to characterize the dropouts in order to check if they are representative of the total sample. However, due to the anonymity of the data, we were unable to do this.

The subjects in the control group are receiving less desirable treatments. As the natural underdog, they might be motivated to reduce or reverse the expected outcome of the experiment. This threatens the comparison in development time between both groups. This effect is expected to be more present if we had been comparing the security of the resulting code, but we did not do this. Moreover, we took the necessary precautions to avoid that the control group was aware of being in a less desirable situation, such as leaving them unaware of what the Sensei tool looks like, thus leaving them unaware of it being disabled for them.

B.9.3. Construct Validity We collected information about the time spent resolving issues as the time between introducing the violation and removing it. However, during this time window the subjects might still be working on the functionality of the code instead of its security. Since these two tasks are mostly interleaved it would be nearly impossible to precisely asses the two times separately. Hence our focus on the increase in total development time as an additional measurement.

The subjects were aware that they were participating in an experiment. This in itself may make the subjects more receptive to its feedback.

The subjects were allowed to use any resource they desired to complete the task. This factor may influence the results, because better resources could help in completing the programming task faster or with less security issues.

The subjects might try to figure out what the purpose and intended result of the experiment is. They are likely to change their behaviour based on their guesses about the hypotheses. For this reason we did not disclose to the participants whether or not they were part of the control group or the test group. But it is likely at least the control group would realise their role in the experiment after not receiving feedback from the tool for a while. This does not influence our results about the interaction with the tool since the control group does not interact with it. The test group is less likely to realise their role in the experiment, but the realization is more likely to cause an effect.

Some people are afraid of being evaluated. A form of human tendency is to try to look better when being evaluated, this could influence how the test group interacts with the tool. It is possible that the subjects would ignore markings more often if they were not being evaluated.

B.9.4. External Validity All students come from the same college and the same bachelor's program. It is possible that subjects from a different college or program might result in different performance while completing the development task.

The subjects were not trained or experienced in the use of the treatment. It is possible that developers with more experience with security tools in general, or specifically Sensei, behave differently when interacting with the tool.

Since all subjects were tasked to develop a web application using Java JSP, the findings might not relate to development in general. The findings might not apply to development of other types of software, or when using other languages, or frameworks.

The subjects mostly lack professional experience, most of them only having done internships. It is possible that developers with more professional development experience behave different.

C. TRIAL WITH CLIENT

In Section 6.5 where we discuss the impact of Sensei on the codebase we mention findings from a trial with a client. During the development of the plugin we have had multiple trials with interested companies, all of which are currently still paying clients. For the report we have chosen this trial in particular due to the transparency of the client's internal process. In this appendix we discuss the trial in more detail.

C.1. Goal and Research Questions

The *goal* of the trial is for the client to observe the effects of the Sensei IDE plugin on its development process. The *purpose* is to help the client decide whether or not it is worth to purchase licenses for the Sensei IDE plugin. The *quality focus* is on the time and money saved by detecting possible vulnerabilities early. The client aims at better estimating the return on investment of the potential purchase. During the trial they can both collect some objective data on the amount of vulnerabilities prevented, as well as collect opinions from the application security team and the developers involved in the trial.

C.2. Subjects

The client is a large bank included among the top 25 banks of the world as listed on wikipedia (https://en.wikipedia.org/wiki/List_of_largest_banks). The subjects were a group of five full-time developers selected by the client for their security knowledge. The tool was also given to an employee responsible for application security to help evaluate the trial. This employee was our main contact during the trial period.

C.3. Tasks

The subjects are part of teams developing and maintaining the web and mobile applications of the client. They were developing in either IntelliJ IDEA or Android Studio. During the trial period they continued their daily responsibilities as usual, reporting periodically to the application security expert on their impressions of the tool.

C.4. Treatment

The developers were given two sets of rules, one for general Java applications, and one for mobile Android applications in particular. The ruleset for general Java applications was developed internally by our developers in cooperation with the application security expert. They advised what they wanted to achieve from the developers with the tool, and we created rules to enforce this. The second ruleset was also developed by us and was based on the official Android developer guidelines (https://developer.android.com/). All of the rules in this set had scopes so they would only be active when the developer is working on an Android project.

C.5. Post-Trial Information Gathering

The client did not share their code nor their Sensei events file. Our contact was given the ability to view the summary of the Sensei events file in the form of an update to the Sensei plugin that enables them to view the statistics on each device. Our contact at the company evaluated these and shared some of their insights as well as opinions from the subjects themselves.

C.6. Results

They reported that during the trial over 200 markings were found that were legitimate markings that could lead to vulnerabilities. With the majority of these present in legacy code, they were potentially vulnerabilities already in production. The two most common categories were mentioned as being tapjacking and sensitive information leakage (mostly caused by leaking stack traces).

The subjects reported the tool as very useful and not too intrusive when working on new code. They also reported improving their security knowledge, driven by the markings from the plugin.

After the trial, the client chose to extend their current licenses and purchase additional ones.

C.7. Threats to validity

There are many threats to the validity of conclusions drawn from the findings of this trial. We have no detailed knowledge or control over the task, the subjects, the time, or indeed over any other aspect of the trial. We are unable to account for any noise in the metrics or any conditions that limit our ability to generalize the results. For this reason we make no attempts at interpreting the results from this trial, we only report the findings as they were reported to us.