An Approach to Secure Legacy Software Systems

Stefanie Jasser[1,2] and Jonas Kelbert[2]
[1]*Department of Informatics, University of Hamburg, Hamburg, Germany*
[2]*akquinet AG, Hamburg, Germany*

Keywords: Software Security, Code Cleansing, Security Refactoring, Vulnerability Mitigation, Flaw Prioritization.

Abstract: When analyzing legacy software for security huge result lists may be generated. These lists may contain more than 1,000,000 potential vulnerabilities. In this paper, we propose an approach to secure such legacy systems: we define a process to systematically assess and process potential vulnerabilities using contextual system knowledge. The process is complemented with tool-supported technical measures to actually mitigate the vulnerabilities and code injection. The approach allows to efficiently repair vulnerabilities in legacy systems while ensuring system availability for critical systems using a safe go-live technique. We evaluate our approach by an industrial case study to show the applicability and flexibility of our code security cleansing approach.

1 INTRODUCTION

Developing a secure software system is a challenging task. Like for other quality attributes, achieving security in large software systems is determined by the software architecture: Usually, code level vulnerabilities are easier to remedy and less crucial to a system's security than its architectural security measures. Several tools exist for analyzing security on the code-level. Some work also deals with architecture level vulnerabilities (Jasser, 2019; McGraw, 2006).

However, for legacy systems security analysis tools on both, the architectural and the code level, often provide huge lists of potential vulnerabilities, e. g. about 800,000 in our case study. Few work exists on systematically fixing vulnerabilities from such analysis results, e. g. taking the effect on the system's overall risks into account when determining measures. Existing approaches lack consideration of the system and vulnerability context. Instead, software vulnerabilities are often remedied non systematically and sub-optimally for their specific context. Based on this, we identified three issues in existing approaches for vulnerability mitigation in legacy systems:

RQ 1: How Can Organizations Deal with Large Quantities of Analysis Results? Processing huge vulnerability lists causes an enormous effort: Not all vulnerabilities can be mitigated. To deal with large quantities of vulnerabilities, organizations need a systematic process to reduce the risks with reasonable effort. In this paper, we provide a methodology that defines a systematic process on assessing, prioritizing and mitigating vulnerabilities (see Section 2).

RQ 2: How Can Vulnerabilities Be Fixed Systematically? Mostly, there several mitigation strategies can be used to fix a vulnerability. Choosing the best option for security and cost-benefit reasons is difficult. In Section 2 we present our funnel approach that supports project teams in narrowing vulnerability lists based on fixing strategies.

RQ 3: How to Ensure System Availability During Mitigation? Fixing vulnerabilities in legacy systems is a complex task: unlike refactoring, fixing vulnerabilities changes the system behavior, i. e. the modifications may also change the functionality. The mitigation must therefore be tested before applying them on the live system. Introducing an additional logging phase provides further reliability. This way, an organization can take appropriate measures to ensure the system availability before putting a mitigation into productive operations. For safe go-live we use our soft cleansing approach (s. Section 3).

To address these issues, we provide a systematic and practicable approach to deal with large lists of potential vulnerabilities in legacy systems. This approach is a framework for individual adaptation.

2 CLEANSING APPROACH

Our approach bases on provided vulnerability analysis results. It defines a systematic and safe way of

Jasser, S. and Kelbert, J.
An Approach to Secure Legacy Software Systems.
DOI: 10.5220/0008902802990309
In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pages 299-309
ISBN: 978-989-758-399-5; ISSN: 2184-4356

mitigating huge lists of potential vulnerabilities. The approach may be used with vulnerability lists from different sources, i. e. it is not limited to a specific security analysis approach.

In the following Section, we list the requirements, we found prior to the conceptual design of our cleansing approach. Based on these requirements, we designed an approach for fixing software vulnerabilities. The concept comprises two aspects of a security cleansing: first, a systematic process and, second, adequate technological and tool support.

2.1 Requirements for a Mitigation Process Approach

For our cleansing approaches we found the following requirements prior to the conceptual design:

Req 1: Respect Individual Security Requirements. Different enterprises have different security guidelines and requirements. They even differ between software projects. The approach should respect and respond to these varying security requirements.

Req 2: Minimize (Manual) Code Changes. Code modifications often have unexpected impact on a software system. To minimize side effects, our approach should prefer solutions that avoid code modifications.For vulnerabilities that need to be fixed through code modifications, the modifications should be done tool-supported if possible.

Req 3: Safe Go-live (Availability of Operational Environment). Mitigating changes may reduce the availability of a system: even correctly implemented changes usually restrict the accessed functionality. Users may not be able to use functionality they used prior to the mitigation. The approach must provide a mechanism to avoid such availability limitations.

Req 4: Optimize Effort-benefit-ratio. To be feasible, the cleansing approach must have a reasonable effort-benefit-ratio. The workload has to be defined based on a systematic prioritization of the overall total vulnerability list. Additionally, the approach should automate as many solutions as possible.

Req 5: Context-sensitive Assessment of Vulnerabilities. Vulnerabilities exist within a software system. The context is essential for choosing a good repair solution. The vulnerability assessment must consider this context of a vulnerability and the system.

Req 6: Meet Individual Quality Alignments. Beyond individual security requirements, enterprises follow different quality and coding guidelines. Although, the approach should be automated where possible, it must meet these quality requirements.

Req 7: Flexibility Regarding Enterprise Processes and Heartbeat. Cleansing projects take place within a company's heartbeat: we need a flexible approach that fit's into company's processes and schedules.

Req 8: Auditability of Process and Documentation. Software security cleansing is required for the overall compliance and security of companies. Hence, the process, decisions made and all documents must be audit compliant including appropriate documentation.

2.2 The Process

Our approach leads to a systematic handling of huge vulnerability lists. It aims to minimize code level mitigation when reducing a system's security risks. To achieve this, we a) identify false positives and vulnerabilities that only cause minor risks, b) do as much mitigation beyond code level as possible and c) minimize manual solutions, if code-level fixing is necessary. This funnel approach is presented in Figure 2. Figure 1 shows the project method we suggest.

Step 1: Project Initialisation and Cleansing Preparation

A code cleansing project starts with a workshop to define the scope of the project: the workshop includes sessions on specific protection objectives, a selection of security rules to check and the specification of an individually acceptable risk level. Through this workshop we meet requirement no 1. Based on the rules in scope, the contextual knowledge needed can be specified. The contextual knowledge on the software system is used to select the solution type for each vulnerability. It may be necessary to collect these data first, if this has not been done prior to the project.

In parallel, the tools to support mitigation should be installed. Furthermore, the presentation of the cleansing method to responsible persons and necessary approvals should be obtained in the meantime.

Step 2: Unused Code Decommissioning

The steps 2, 3 and 4 are interchangeable: their order can be changed or they can be executed in parallel, if the usage data on a system are available prior to the project. Otherwise, they have been collected in Step 1 as a part of the context data.

Unused code objects and fragments have not been used on the live system within a certain period of time. This may usually be 6 or 13 month. We use this usage

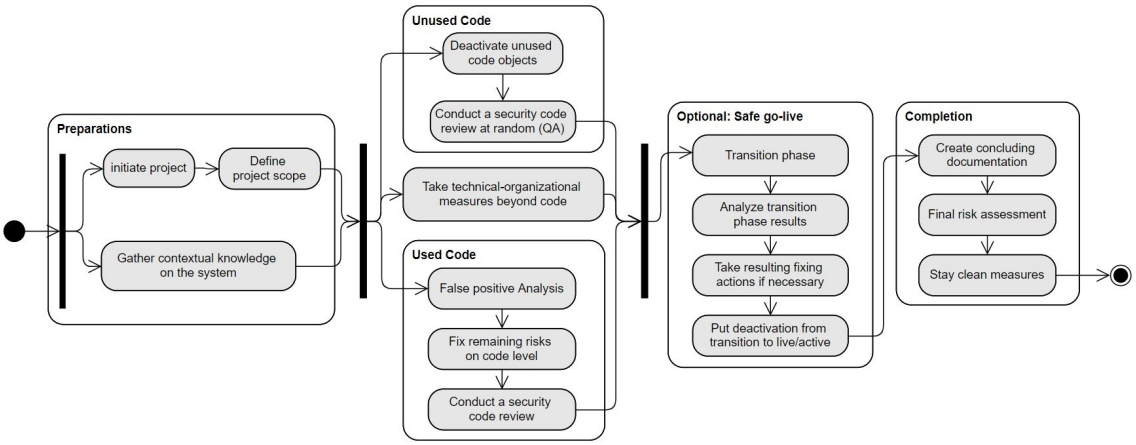

Figure 1: Project method.

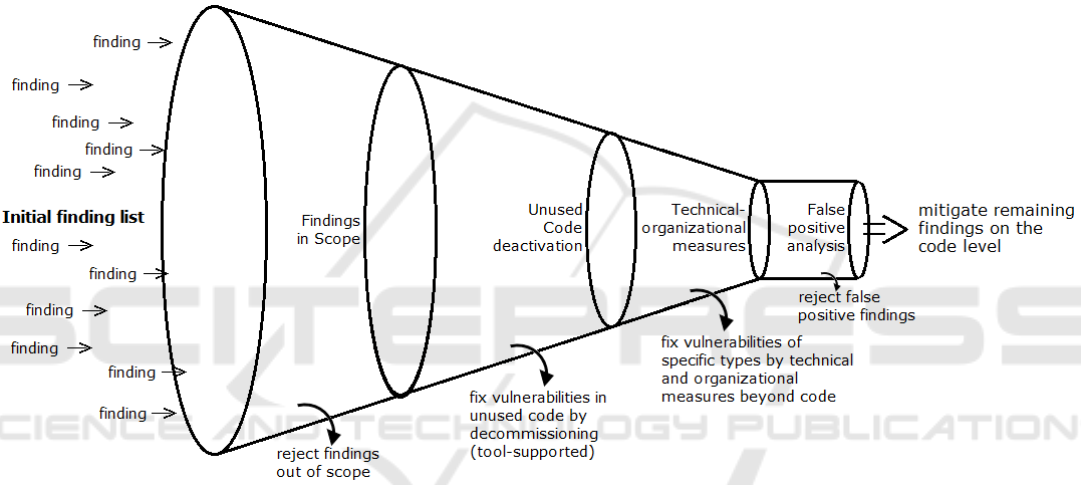

Figure 2: The funnel approach to minimize (manual) code modifications.

statistics to get rid of inactive code and it's vulnerabilities Besided mitigation of its vulnerabilities, we do not need to put maintenance effort into this code any more. However, most developers resist to actually delete code in the first step: We use an easy to reverse decommissioning approach. The decommissioning is done by code injection and control mechanisms that allow to deactivate the mitigation logic without further code changes (see Section 2.3).

Step 3: Technical-organizational Mitigation

Changing source code is risky, inter alia, due to unpredictable impacts on the system. We reduce the number and impact of code modifications due to their fault proneness: Concerning vulnerabilities in active code, we prefer technical or organizational measures that supersede code modifications. For instance, many frameworks provide functionality to limit the paths a user may access from an application and, thereby, reduce the risk of directory traversal attacks. Before making potentially unnecessary code modifications, appropriate configuration and customization of the system and its building blocks should be ensured.

It depends on the software system's context, which technical or organizational measures can be applied. For instance, this is affected by the technologies used and the project's basic conditions.

Step 4: Mitigation through Code Modifications

To invest no unnecessary effort in code modifications, we first conduct an analysis of the findings in scope to identify false positives and findings that do not need to be mitigated due to low risk.

Although we try to avoid code modifications, they are necessary sometimes: in these cases, we intend to automate the code modifications. Each vulnerability may be caused by multiple code patterns. We use the patterns to identify the code fragments to mod-

```
if(currentUser
        == "jdoe"){
    doSomething();
}
```

```
switch(currentUser){
        case "jdoe":
        case "jmodaal":
        case "zsan":
            doSomehting();
            break;
        default:
            break;
}
```

Figure 3: Two exemplary vulnerability pattern for hard-coded username checks.

```
try {
    AccessController.checkPermission(
                        currentUser,
                        new Permission(
                            "someFunction",
                            "access"));
    doSomething();
} catch (AccessControlException e) {
    throw AccessDeniedException(
                        "Subject has no access");
}
```

Figure 4: Mitigating code modifications for both examples.

ify. Based on the vulnerability type and the identified code pattern, the necessary actions are determined, e. g. code injections or the creation of table entries.

Figure 3 shows two exemplary code patterns of hard-coded username checks: an if and a switch case example that check the current username against hard-coded values. The result influences the control flow and, thereby, the system behavior. In this example, the code would be modified as showed in Figure 4: the hard-coded username checks would be replaced by checking the user's authorization. The hard-coded users must be granted the permissions.

If a vulnerability's code pattern can be mitigated using well-defined actions, these actions can be done with tool-support. However, the mitigation should not be done in batch mode but with a control mechanism: each modification should be checked by a developer. Thus, the mitigation is more efficient and less fault-prone, because the developer does not need to change the code manually. Hence, the developer will mitigate more findings per day in this semi-automated mode as in a completely manual mode. Additionally, the solutions will probably cause less faults.

Some vulnerabilities have to be mitigated manually due to their complexity or severity: such vulnerabilities can only be fixed by changing the fundamental logic of the program. For instance, this is the case if code is uploaded and executed dynamically. Such functionality often exists to repair incoming data during run-time. However, an attacker may use this functionality, to upload and execute malicious code.

Step 5: Safe Go-live

Applying the safe go-live approach to the cleansing project is optional. We recommend it for software systems that require high availability.

The approach makes the go-live safe by setting the mitigation live gradually: It adds one transition phase at the least. During this transition phase, the previous and the new mitigation logic are executed. If the results diverge, this divergence is logged using an auditable log. As the mitigation will usually narrow the access to or generality of a functionality, most divergences are about a user who could access a functionality prior to the mitigation and cannot access it any more using the new logic. Finally, the original logic is executed that applied prior to the mitigation.

Dependent on the vulnerability or chosen solution, there may be additional transition phases implemented. The safe go-live approach is described in more detail in Section 3.

Step 6: Project Completion

To complete the project, the documentation must be finalized in an auditable way. Ideally, all major steps and decisions have been aligned with the auditor directly throughout the project.

Partly, the auditable documentation has been created throughout the project such as logs during the transition phase or the methods and techniques used for mitigation. However, the final result must be documented and presented at the project end. This includes assessing the remaining risks that have been accepted due to effort-benefit ratio.

Beyond auditable documentation, measures for staying clean must be defined: these may be technical as well as organizational measures. Usually, the measures include a security awareness and secure code training and an adapted software development process. The process stipulates frequent tool-based security analysis or secure code reviews.

2.3 Tool-supported Mitigation

The tool we developed intends to support and partially automate the mitigation of vulnerabilities. It is intended to be used by developers.

2.3.1 Guided Code Vulnerability Mitigation

Recurring vulnerability patterns are often remedied with identical solutions. We developed a tool that uses code injection techniques to repair such vulnerabilities. For tool-supported fixing, there are pre-defined

vulnerability patterns. Unknown vulnerability patterns cannot be fixed using the tool. This applies to unknown variants of known vulnerabilities, too. If the tool finds a known vulnerability pattern near to the line in code at which the vulnerability occurs, it selects the assigned solution for mitigation. The tool repairs the vulnerability by code injection. Usually, the code injection is not done in a batch but in a guided mode: In the guided mode, the workload is processed step by step. An editor shows the suggested solution that would be applied. If the tool inserts or modifies table entries, this is displayed, too. For instance, table entries are inserted into the safe go-live control tables, if the safe go-live approach is used. The developer may accept, reject and skip or manually modify a solution. The safe go-live approach should be used for all code injections by the tool.

All modifications that are made by the tool have to be logged in an auditable way: not only the original and the modified code but also the executing user is logged. Additionally, every manual adaptation of the suggested solution is logged.

2.3.2 Unused Code Decommissioning

Unused code usually contains as many vulnerabilities as used code. By decommissioning unused code, these vulnerabilities are remedied with little effort. Decommissioning is done by injecting code into the inactive code objects or fragments and creating control table entries on the database. The table entries determine the fixing status, the message to be shown and the mitigation code's actual behavior for each mitigated vulnerability. It is used to apply the safe go-live approach. These information are queried by the injected code. E. g., the fixing status may be "logging". In this case, the previous and the modified logic are executed on trial. If the results differ, the difference is logged. However, the previous logic is actually executed in the end.

Beyond mitigating the vulnerabilities in unused code, the decommissioning improves the system's maintainability and portability. This is due to the code must no longer be considered during maintenance or porting.

3 SAFE GO-LIVE: SOFT CLEANSING

Availability is an essential requirement for many software systems. There are several ways to unintentionally limit the availability through the cleansing activities: Code modifications have the inherent risk to cause faults in the system. This may be due to wrongly implemented security fixes and modified functionality or due to unexpected impacts on the system. Yet, correct modifications may limit the availability, too: For instance, in case of a missing authorization check, the straightforward fix is the introduction of a reasonable authorization check. This probably limits the availability as users may not be able to access the functionality any more. However, some of these users may be justified to access the functionality. Such users may not have the permissions because the missing permissions did never attract attention due to the lack of authorization checks.

For software systems that are required to be highly available we developed a safe go live approach for our cleansing projects: the soft cleansing approach. The soft cleansing approach puts the mitigating code changes into productive operations step-by-step.

As Figure 5 shows, the mitigating code is implemented in the first step. The code is extended with a control logic that enables the gradual go-live. The implementation may be done using automated code injection or by hand. Referring to the example presented in Figure 4, the code would be extended as in Figure 6: The control logic queries the underlying control tables that hold the soft cleansing status. If the cleansing is in transition phase, the injected authorization check is performed. If the access is denied, the program executes the prior logic indicated by `doSomething()` and logs the discrepancy between prior and mitigation logic. The opposite case – a user had no access before mitigation and can now access the functionality – can not happen as we had a missing authority check before. When the mitigation is set live finally (cleansing status `ACTIVE`), the new logic actually refuses access and throws an `AccessDeniedException`.

Soft cleansing is optional. If it is applied to a project depends on the trade-off decisions regarding availability and other protection goals.

4 FIELD STUDY

4.1 Initial Situation

We conducted our evaluation project with an industrial partner in the technology sector. The project was part of a security initiative. Initially, our industrial partner conducted a vulnerability analysis on their software systems. The analysis resulted in more than 800,000 findings, i. e. potential vulnerabilities. Findings may be architecture-level or code-level vulnerabilities. Analogously to security architecture viola-

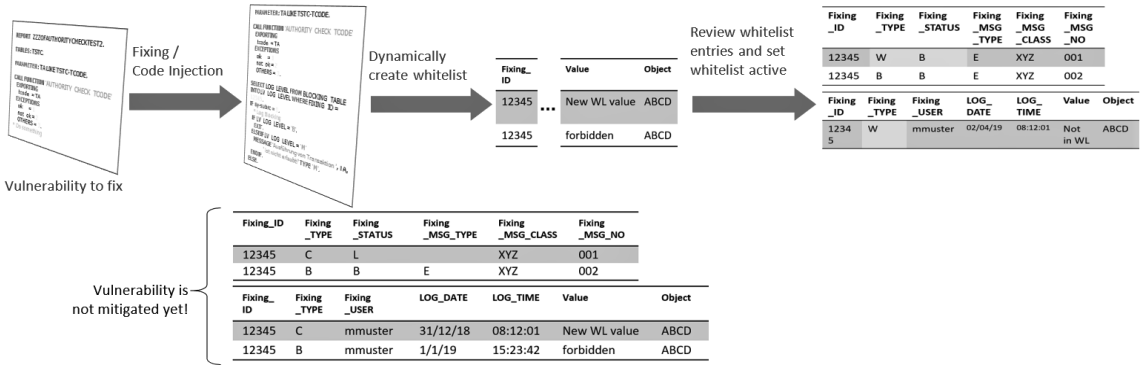

Figure 5: Safe go-live process.

```
1  Long fixId = 12;
2  CleansingStatus status = AccessMitigationController
3                              .getFixInformation(fixId)
4                              .getCleansingStatus();
5  ILogDetail logInfo = new MitigationLogDetail(fixId);
6  boolean performLogic = false;
7  try {
8      AccessController.checkPermission(currentUser,
9                       new Permission("someFunction",
10                                     "access"));
11     performLogic = true;
12 } catch (AccessControlException e) {
13     if(status == CleansingStatus.ACTIVE) {
14         throw AccessDeniedException(
15                 "Subject has no access");
16         } else {
17             performLogic = true;
18             logInfo.setMitigationLogicResult(e);
19         }
20 } finally {
21     if(performLogic){
22         doSomething();
23     }
24     if(logInfo.containsDivergencies()
25          && status != CleansingStatus.NO_LOGGING){
26         logger.logAccessDeclined(logInfo);
27     }
28 }
```

Figure 6: Mitigating code modifications including soft cleansing logic.

tions, findings indicate vulnerabilities but may also be false positives. The partner estimated that an experienced software engineer may fix 4 vulnerabilities per hour at average. This results in an overall effort of about 25,000 man-days.

Our industrial partner aimed to minimize cyber security risk. To achieve this goal, we applied our code security cleansing approach to the system. Due to company guidelines, we had to fit our approach in an agile project method that was similar to SCRUM.

4.2 Prototype

We implemented a prototype to apply our approach to the industrial project. Its essential part is the semi-automatic vulnerabiltiy fixing that is realized through a fixing wizard: besides other information, it proposes a solution by code injection that is highlighted in the editor. The developer is responsible for reviewing the solution. If the developer assesses the solution inappropriate, he can either edit it or skip the vulnerability . Every modification is logged auditably: i. e. The logs are protected from modification or data loss and are in a readable and machinable format.

As the executing user of the tool takes responsibility and it needs software and security engineering knowledge to assess the proposed solutions, the user should always be a software developer. The developer must have security knowledge to enable well-grounded assessment.

4.3 Applying the Cleansing Approach

We applied the cleansing approach to our partner's most important software system. It controls crucial production lines. At the beginning, we had about 800,000 potential vulnerabilities, which resulted from a tool-supported code security analysis. The first step in coping with this huge task list is to define the potential vulnerabilities to be considered in the project: We held a workshop to define the project scope, i. e. to prioritize and determine the subset of findings to fix. Since the concerned software system controls production lines, high availability is mission critical. Hence, availability is the most important protection objective to achieve. However, the considered software system contains sensitive information such as information on products, innovations and internal processes of the technology company. The confidentiality of these sensitive information's must be protected. This is the second most important protection objective as agreed in the workshop.

Based on the protection goal priorities, we assessed the vulnerability types from the analysis tool's documentation: e. g., the upload and execution of code fragments may have an enormous negative impact on the system's availability and confidentiality. Hence, this vulnerability type is assessed highly critical. Other vulnerabilities may have write or only read access, such as SQL injection vulnerabilities. Usu-

ally, we evaluate write access more critical than read access. To further narrow the scope we only consider findings that are likely to have an adverse effect on the software system. This assessment is done by the analyzing tool based on different parameters such as if the vulnerability can be affected by user input.

In parallel, we gathered usage data from the live systems to identify unused code objects. Through decommissioning of unused code we mitigated about 70 % of all findings (ca. 570,000 findings).

We split the remaining list of potential vulnerabilities based on their fixing strategies, i. e. if they may be fixed by technical-organizational measures beyond code and within the source code. For instance, there were existing mechanisms to restrict the access of underlying system paths via authorizations. The system automatically checks these authorizations for every system path access. To mitigate directory traversal vulnerabilities, we redesigned and customized the authorization concept considering this aspect in collaboration with our industrial partner.

Subsequently, the developers analyzed all findings that were not fixed before. They all had knowledge and experience in software security. All potential vulnerabilities were assessed regarding their damage potential: they identified false positives and findings with minor risk. The remaining vulnerabilities were critical and must be mitigated by code modifications.

Most code-level mitigation was done automatically: using the guided mode described above the tool proposes a solution to the developer. The developer reviewed the solution before it is finally applied. The tool logs each code injection in an auditable way. In our project the proposed solutions have rarely been modified but have mostly been injected as proposed. However, we had some vulnerabilities that had to be mitigated individually. Mostly, these vulnerabilities or their context are complex. Therefore, the logic has to be rebuilt in a secure way to fix such vulnerabilities.

Concluding, we conducted code security reviews on all manual mitigations. Vulnerabilities that have been fixed by injecting a default solution or that have been decommissioned are reviewed at random. Our partner did the functional test.

We discussed the process and decisions with the auditor throughout the project to obtain their approval. Due to their consent, the project and system successfully passed the subsequent audit. On demand of the auditor we defined an overall risk metric that is calculated from the vulnerabilities' criticality and the probability of exploiting them: We defined the criticality of the vulnerability types in relation to each other while the analysis tool provides an estimation if it is very likely, less likely or unlikely that a vulnerability may be exploited. We translated this estimation into a factor that is multiplied with the criticality value per finding. To calculate the system's risk level and it's relative changes over time, we added all finding values within the system. Thus, we were able to show the mitigation effect that our project had on the system. The metric was defined in a joint workshop with representatives of the project team, the risk management, the information security officer, the software development department and the auditor.

4.4 Measures to Stay Clean

We trained a selected group of developers in code vulnerabilities and their mitigation in legacy systems. These developers build multipliers of knowledge within the company: they transfer the knowledge they gained to the whole development team.

Additionally, we introduced technical measures: before functional testing the code is checked for potential vulnerabilities using the analysis tool. There must not be any findings that match the project scope definition. Such findings must be mitigated before functional testing and release of the functionality.

5 DISCUSSION

Based on the application of our approach to an industrial case, we discuss the benefits and potential improvements of our approach in the following. First, we discuss if our approach meets the requirements we specified for it (see Section 2.1). Subsequently, the issues mentioned in the introduction are discussed.

5.1 Field Study Results

Respect Individual Security Requirements (Req 1). Our approach stipulates an introductory workshop. An essential part of this workshop is the scope definition: in our field study, we prioritized the company's individual protection goals. Based on that, we evaluated the criticality of the identified vulnerability types. The mitigation scope can be narrowed or broadened dependent on the stakeholder's needs. Another tailored measure is the safe go-live option: The team may apply it to the project if availability is important or may skip it if an immediate mitigation of vulnerabilities is required instead. By the individual prioritization and scope definition, the approach exactly considers the individual security requirements.

Minimize (Manual) Code Changes (Req 2). Our approach aims to mitigate as many vulnerabilities as

possible beyond code: in our field study, we could mitigate more than 75 % of the overall vulnerabilities and more than 90 % of the vulnerabilities in scope by decommissioning unused code, technical-organizational measures and tool-supported mitigation through code injection. In numbers, this means more than 10,000 critical vulnerabilities in scope and about 605,000 vulnerabilities overall have been fixed without manual code modifications. Despite these results are excellent, they are highly dependent on the concrete system. However, we expect a major effects of these steps in most legacy systems. Subsequently, we conducted the false positive analysis on the remaining vulnerabilities in scope. Due to the initial scope definition, 779 findings had to be analyzed in this step, i. e. 779 vulnerabilities. 371 of these vulnerabilities had to be mitigated manually in the end.

However, through automation of the mitigation, developers may not be sensitized as if they had to mitigate the vulnerabilities manually. Due to the cost-effectiveness, we consider the tool-support valuable. Yet, taking measures to build the developers' awareness is a reasonable supplement.

Safe Go-live (Availability of Operational Environment) (Req 3). We enable the safe go-live by adding another optional phase to our approach. During this phase, divergences in the result of previous and modified logic after mitigation are logged. In this case, the previous logic is executed finally.

As the approach can be controlled using control tables that have each mitigation's safe go-live status, modifications can be reverted easily in case of an emergency situation. There is only a minimal risk to the live system as a result. However, the vulnerabilities are not actually mitigated until the safe go-live status is set to active. This is due to the original code is executed finally during transition phase.

Optimize Effort-benefit Ratio (Req 4). As described above for Req 2, the effort-benefit ratio is optimized by narrowing the project scope and automated mitigation. The cost-effectiveness further increases due to additional steps in the project like conducting a false positive analysis. Due to this methodology, we only put effort into critical vulnerabilities that cause high or very high risks for the system and organization.

As mentioned in Section 4.1, our partner estimated about 25,000 man-days for a straightforward manual mitigation. We were able to reduce the major risks of the system in about 460 man-days.

Context-sensitive Assessment of Vulnerabilities (Req 5). We assessed the vulnerabilities using their runtime context: we gathered information on the usage of code objects from the live systems. Our tool proposes code objects for decommissioning if they were inactive for a specified period of time. As our industrial partner already gathered usage data prior to the code cleansing project, we used the usage data of 15 months in our field study. It includes exceptional usage at year end/beginning or quarter end/beginning.

Additionally, developers consider a vulnerability's context when conducting a false positive analysis: they use the actual context to assess if a vulnerability may be exploited or not.

Meet Individual Quality Alignments (Req 6). Controlling the adherence to organization specific quality guidelines are part of our security code review checklist. Additionally, our fixing suite provides many common security solutions like the introduction of authorization checks or whitelisting mechanisms. By injecting code that uses these components, we enforce common design principles such as component reuse. However, during our field study we found that our prototype may be improved in following organization specific developer guidelines such as naming conventions: the tool should allow to adapt the mitigating code templates that are injected. Otherwise, the injections either do not follow the developer guidelines or they have to be corrected manually.

Flexibility Regarding Enterprise Processes and Heartbeat (Req 7). In our field study, we had to apply our approach to an agile project method. This worked well for the code cleansing approach: The vulnerabilities usually are well-defined and manageable tasks to perform within a sprint. This facilitates reliable estimations after a few sprints. Additionally, agile project methods require short feedback cycles. As our project team worked in parallel to the actual development team, these short feedback cycles proved useful.

Auditability of Process and Documentation (Req 8). To ensure the auditability of the code cleansing project, our prototype wrote logs for each modification. These logs were designed for auditability: they were protected from modifications and data loss.

Additionally, we aligned with the auditors from the beginning. During the first project phase, we presented the project method to the auditors to get their approval. As agreed, we further discussed the intended mitigation measures per vulnerability type with them, before starting the actual mitigation.

However, the documentation of risk reduction had to be done manually throughout the project. We created the overview of mitigated and accepted risks as

well as the risk metric mentioned in Section 4.3 manually. There should be future work on creating a more lightweight, tool-supported documentation.

5.2 Code Cleansing Issues

In the introduction, we mentioned three issues that we identified when dealing with huge lists of vulnerabilities. These issues mainly refer to the lack of systematic methodology and reasonable tool-support for the mitigation of large quantities of vulnerabilities.

How Can Organizations Deal with Large Quantities of Analysis Results? (RQ 1)

This issue is covered by the systematic process and the tool-supported mitigation of vulnerabilities which minimizes manual effort: Through the scope definition we were able to exclude non-critical findings from the project which had an enormous effect in the amount of work. Through measures beyond code and tool-supported code injection, we enabled developers to mitigate more than 90 % of the remaining findings within a few days. It is affected by the requirements Req 1, 2 and 4 or their fulfillment, respectively.

How Can Vulnerabilities Be Fixed Systematically? (RQ 2)

The systematic mitigation is looked at in the requirements Req 1, 2 and 5: Similarly to RQ 1, we met this issue with our methodology which defines a systematic process for the mitigation of vulnerabilities. Additionally the code injection functionality leads to a consistent use of mitigation measures throughout the code. This only excepts for a few vulnerabilities that had a particular context or that were too complex for these solutions.

How to Ensure System Availability During Mitigation? (RQ 3)

The system's availability may be ensured by applying the safe go-live approach presented in Section 3. Req 3 considers the safe go-live approach. The benefits and drawbacks mentioned for this requirement also hold true for RQ 3. Additionally, Req 6 treats the code modifications' quality assurance. Through code reviews we enforce the four eyes principle. Reviewers should pay attention to modifications' unexpected effects on the system availability.

6 RELATED WORK

In the field of software security, a lot of work has been done recently. However, we didn't find any approach that handles huge vulnerability lists systematically as ours that combines assessment, prioritization, automatic repair and availability assurance.

Security Analysis. (Li et al., 2013) present a dynamic approach to detect vulnerabilities: They analyze backwards traces to identify abnormal results. (Almorsy et al., 2012) performs security scans on OCL definitions for web applications and interfaces (Almorsy et al., 2012). Other works deal with security conformance checking to identify fundamental vulnerabilities in software systems (Jasser, 2019; Abi-Antoun and Barnes, 2010). (Sametinger, 2013) gives a short discussion on the increasing security awareness and points out security flaws in the software architecture as one major source of vulnerability risks.

Systematic Vulnerability Mitigation. Systematic refactoring processes that do not affect the code behavior are proposed in (Mortensen et al., 2012; Rizvi and Khanam, 2011). Both approaches use aspect oriented programming (AOP) techniques. May authors discuss the need of prioritization and fixing strategy knowledge to handle a large quantity of flaws (Tornhill, 2018; Li et al., 2017; Mair et al., 2014; Herold and Mair, 2014; Mair and Herold, 2013).

Tool-supported approaches aim to reduce the effort of software vulnerability mitigation: several work has been done on automatic code modifications (Xing and Maruyama, 2019; Anand and Ryoo, 2017; Durieux and Monperrus, 2016; Hansson, 2015; Qi et al., 2012; Le Goues et al., 2012). (Gazzola et al., 2019) give an overview of generate-and-validate and semantic driven mitigation approaches. In (Tang and Huang, 2016) the authors present a soft-error mitigation by considering a combination between functional unit configuration and instruction flow. Several authors provide results on mitigation strategies that are based on semi-automatic analysis of vulnerabilities (Brunil et al., 2009) or API specific patterns (Nielebock, 2017). None of the approaches try to minimize fault prone code changes, e. g. by considering countermeasures beyond code or unused code.

Security Engineering. There is a wide range of discussions, papers and publications related to secure software (Gazzola et al., 2019; Xu and Peng, 2016; Gao et al., 2016). Due to the fast development of software automatic detection and fixing becomes more and more important. The fixing can also be supported by machine learning (Tommy et al., 2017).

7 CONCLUSION AND FUTURE WORK

In this paper, we presented an approach for systematically dealing with huge vulnerability lists that may result from security analyses. Therefore, our approach defines both a methodology and a technical approach. While the methodology specifies how to execute a code cleansing project, the technical approach provides support in mitigating the vulnerabilities beyond code or at least with minimal manual effort. The approach has been applied to an industrial field study for evaluation. Therefore, the technical approach has been implemented prototypically.

Currently, we conduct additional field studies for a more reliable evaluation of our approach. We intend to iteratively optimize our tool-supported mitigation through code injection based on the field studies. One issue we want to improve is the adaptation to individual development guidelines such as naming conventions. Therefore, we need to provide a possibility to edit the code to inject for mitigation.

REFERENCES

Abi-Antoun, M. and Barnes, J. M. (2010). Analyzing security architectures. In Pecheur, C., Andrews, J., and Di Nitto, E., editors, *Int. Conf. on Automated Software Engineering*, pages 3–12. ACM.

Almorsy, M., Grundy, J., and Ibrahim, A. S. (2012). Supporting automated vulnerability analysis using formalized vulnerability signatures. In *27th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 100–109.

Anand, P. and Ryoo, J. (2017). Security patterns as architectural solution - mitigating cross-site scripting attacks in web applications. In *Int. Conf. on Software Security and Assurance*, pages 25–31.

Brunil, D., Haddad, H. M., and Romero, M. (2009). Security vulnerabilities and mitigation strategies for application development. In *6th Int. Conf. on Information Technology: New Generations*, pages 235–240.

Durieux, T. and Monperrus, M. (2016). Dynamoth: Dynamic code synthesis for automatic program repair. In *IEEE/ACM 11th Int. Workshop in Automation of Software Test*, pages 85–91.

Gao, F., Wang, L., and Li, X. (2016). Bovinspector: Automatic inspection and repair of buffer overflow vulnerabilities. In *31st IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 786–791.

Gazzola, L., Micucci, D., and Mariani, L. (2019). Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67.

Hansson, D. (2015). Automatic bug fixing. In *16th Int. Workshop on Microprocessor and SOC Test and Verification*, pages 26–31.

Herold, S. and Mair, M. (2014). Recommending refactorings to re-establish architectural consistency. In Avgeriou, P. and Zdun, U., editors, *8th Europ. Conf. on Software Architecture*, volume 8627 of *Lecture Notes in Computer Science*, pages 390–397. Springer.

Jasser, S. (2019). Constraining the implementation through architectural security rules: An expert study. In *20th Int. Conf. on Product-Focused Software Process Improvement*.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Li, H., Kim, T., Bat-Erdene, M., and Lee, H. (2013). Software vulnerability detection using backward trace analysis and symbolic execution. In *Int. Conf. on Availability, Reliability and Security*, pages 446–454.

Li, X., Chang, X., Board, J. A., and Trivedi, K. S. (2017). A novel approach for software vulnerability classification. In *Annual Reliability and Maintainability Symposium*, pages 1–7.

Mair, M. and Herold, S. (2013). Towards extensive software architecture erosion repairs. In Drira, K., editor, *7th Europ. Conf. on Software Architecture*, volume 7957 of *Lecture Notes in Computer Science*, pages 299–306. Springer.

Mair, M., Herold, S., and Rausch, A. (2014). Towards flexible automated software architecture erosion diagnosis and treatment. In *Working Int. Conf. on Software Architecture Companion Volume*, WICSA '14 Companion, pages 9:1–9:6, New York, NY, USA. ACM.

McGraw, G. (2006). Software security: Building security in. In *17th Int. Symposium on Software Reliability Engineering*, pages 6–6.

Mortensen, M., Ghosh, S., and Bieman, J. (2012). Aspect-oriented refactoring of legacy applications: An evaluation. *IEEE Transactions on Software Engineering*, 38(1):118–140.

Nielebock, S. (2017). Towards api-specific automatic program repair. In *32nd IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 1010–1013.

Qi, Y., Mao, X., and Lei, Y. (2012). Making automatic repair for large-scale programs more efficient using weak recompilation. In *28th IEEE Int. Conf. on Software Maintenance*, pages 254–263.

Rizvi, S. A. M. and Khanam, Z. (2011). A methodology for refactoring legacy code. In *3rd Int. Conf. on Electronics Computer Technology*, volume 6, pages 198–200.

Sametinger, J. (2013). Software security. In *20th IEEE Int. Conf. and Workshops on Engineering of Computer Based Systems*, pages 216–216.

Tang, L. and Huang, Z. (2016). A method for issue queue soft error vulnerability mitigation. In *17th IEEE/ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 443–450.

Tommy, R., Sundeep, G., and Jose, H. (2017). Automatic detection and correction of vulnerabilities using machine learning. In *Int. Conf. on Current Trends in*

Computer, Electrical, Electronics and Communication, pages 1062–1065.

Tornhill, A. (2018). Prioritize technical debt in large-scale systems using codescene. In *IEEE/ACM Int. Conf. on Technical Debt*, pages 59–60.

Xing, X. and Maruyama, K. (2019). Automatic software merging using automated program repair. In *IEEE 1st Int. Workshop on Intelligent Bug Fixing*, pages 11–16.

Xu, D. and Peng, S. (2016). Towards automatic repair of access control policies. In *14th Annual Conference on Privacy, Security and Trust*, pages 485–492.