Ingvild Løes Nilsson

Self-Healing after Security Incidents

June 2020

Master's thesis

Master's thesis 2020 Ingvild Løes Nilsson

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Information Security and Communication
Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Self-Healing after Security Incidents

Ingvild Løes Nilsson

Communication Technology
Submission date: June 2020
Supervisor: Danilo Gligoroski
Co-supervisor: Felix Leder

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

NTNU – Trondheim
Norwegian University of
Science and Technology

Self-Healing after Security Incidents

Ingvild Løes Nilsson

Submission date:	June 2020
Responsible professor:	Danilo Gligoroski, ITEM
Supervisor:	Felix Leder, NortonLifeLock

Norwegian University of Science and Technology
Department of Telematics

Title: Self-Healing after Security Incidents
Student: Ingvild Løes Nilsson

Problem description:

Over the past years, both the frequency of cyberattacks, as well as the sophistication of the attackers, have increased. Every day, enterprises are struck by digital espionage, data loss and system hijacking. With the growing threat landscape and highly complex computer systems, the maintenance and administration of these structures have become unmanageable for humans. As the threat landscape widens with the development of new technology, it becomes alarmingly difficult to control all risks associated with these new advancements. The consequences are demonstrated through consistent, successful cyberattacks launched towards vulnerable systems. Hence, the ability to identify the shortcomings of these systems is deficient. The necessity for automated approaches discovering susceptibilities and weaknesses in infected, complicated infrastructures is greater than ever.

Since IBM released its manifesto in 2001, claiming that automated software shall be the resolution to the software complexity crisis, the interest in systems possessing certain self-* properties have escalated. If the systems were able to monitor, optimize and heal by themselves without human intervention, the benefits would be tremendous. A self-healing system would be able to detect the error, bug or vulnerability which allowed an intrusion to happen, as well as removing this susceptibility autonomously. Even though the excitement towards autonomic computing is present, the research area is fairly hard and new, and therefore limited.

Motivated by the increasing interest in the topic, the potential benefits of such self-* systems and the scarce research within the field, this master thesis will explore the status of today's self-healing systems and examine self-healing techniques.

The main tasks of the thesis include:

- Explore literature related to self-healing systems, discuss the self-healing techniques used in the previous work and what the shortcomings of such systems are.
- Investigate how to create an autonomous self-healing system with the use of Docker containers and vulnerability tools.

Responsible professor:	Danilo Gligoroski, ITEM
Supervisor:	Felix Leder, NortonLifeLock

Abstract

As services become increasingly digitalized, the attack surface is also expanding for criminals operating digitally. Modern systems have reached a level of complexity that is difficult to maintain and secure; today's solutions with anti-malware programs and perimeter security fail to keep up with hackers' attacks and methods. The need for aware machines that can detect attacks themselves, and correct vulnerabilities in their systems without human interaction emerges. *Self-healing* machines is perceived as a viable option in order to handle the overwhelming complexity of modern digital systems, as well as protecting against malicious attacks.

In this master thesis, the topic *self-healing machines* is explored through a theoretical and a practical approach. The theoretical approach includes a literature study, where existing literature in the field of research is mapped and presented. In the practical approach, a self-healing script is developed. Appropriate tools and platforms are selected, which include a vulnerable network application, an intrusion detection system, and a vulnerability scanner. In addition, five healing techniques are defined. The results from the thesis indicate that the development of systems with self-healing properties is severely complicated. The degree of self-healing is at the expense of non-functional and functional requirements, such as availability and functionality. The uniform healing mechanisms have a greater healing range, but at the cost of other application requirements. The more specified healing functions entailed greater complexity in development, but retained better non-functional and functional requirements. The results from the literature study show that the research topic is still immature, with limited research.

Sammendrag

Ettersom tjenester blir stadig mer digitalisert, ekspanderer også angrepsflaten til digitale forbrytere. Moderne systemer har nådd et nivå av kompleksitet som er vanskelig å vedlikeholde og sikre; dagens løsninger med antiskadevareprogrammer og perimetersikring klarer ikke holde tritt med hackernes angrepsmetoder. Det har utartet seg et behov for bevisste maskiner som selv kan detektere angrep og korrigere sårbarheter i sine systemer uten menneskelig interaksjon. *Selvhelbredende* maskiner blir snart en nødvendighet for å kunne håndtere den overveldende kompleksiteten til moderne, digitale systemer, samt beskytte mot ondsinnede angrep.

I denne masteroppgaven blir temaet *selvhelbredende maskiner* undersøkt gjennom en teoretisk og en praktisk tilnærming. Den teoretiske tilnærmingen inkluderer en litteraturstudie, der eksisterende litteratur innen forskningsfeltet blir kartlagt og presentert. I den praktiske tilnærmingen blir et selvhelbredende skript utviklet. Passende verktøy og plattformer blir valgt ut, hvilket inkluderer en sårbar nettapplikasjon, et inntrengingsdeteksjonssystem og en sårbarhetsskanner, samt fem helbredingsteknikker blir definert. Resultatene fra oppgaven indikerer at utvikling av systemer med selvhelbredende egenskaper er svært komplisert. Grad av selvhelbreding går på bekostning av ikke-funksjonelle og funksjonelle krav, som eksempelvis tilgjengelighet og funksjonalitet. De uniforme helbredingsmekanismene har en større helbredingsrekkevidde, men på bekostning av andre krav til applikasjonen. De spesifiserte helbredingsfunksjonene innebar større kompleksitet i utviklingen, men bevarte til gjengjeld ikke-funksjonelle og funksjonelle krav bedre. Resultatene fra litteraturstudien viser at forskningsemnet fremdeles er umodent, med begrenset forskning.

Preface

This master thesis completes my five years study as a student of Communication Technology at the Department of Information Security and Communication Technology at the Norwegian University of Technology and Science (NTNU). The thesis is written in collaboration with NortonLifeLock.

I would like to thank my supervisors Danilo Gligoroski and Felix Leder for their guidance. I would also like to thank Ina Rekk Bjørnestad for proofreading my thesis, and for being my companion throughout these years in Trondheim.

Trondheim, Thursday 4th June, 2020
Ingvild Løes Nilsson

Contents

List of Figures

List of Tables

Chapter 1 Introduction

With more than 3 billion Internet users as of 2016 [MROO20], the world has never been more digitized. Digital services have gained foothold in people's everyday lives, and have become essential for communication, transport, public services, and more. These services are dependent on secure solutions, but as new technologies are implemented, the threat horizon expands. Attackers grow more sophisticated, and both end users and businesses are being targeted.

Monitoring, identifying and blocking malicious activities towards computer systems are becoming increasingly complicated, while maintaining a vulnerability free-system is nearly impossible. Today's computer systems are highly complex, which is one of the causes of why traditional detection and prevention mechanisms are unable to ensure secure digital infrastructure and platforms. Currently, cyberincidents often require manual inspection, which is neither scalable nor sustainable. Having the machines autonomously recover from cyberattacks by "self-healing," without the need for human intervention would increase the overall efficiency and security of the system.

1.1 Motivation

The approach to handling security incidents have gone through significant changes over the years, and are reflected in the NIST Cybersecurity Framework[1]. It contains five stages involved with preventing, detecting, and recovering from cyberattacks. Whereas identification of risk and threat actors, and means of protecting information systems, were the main research areas some decades ago, detection of anomalies and events, as well as how to respond to unwanted cyber incidents, have become hot research topics over the recent years. The final stage of the framework, which is to recover from cyberattacks, has minimal research as of today in contrast to the prior four stages. Automated self-recovery and self-healing are fields with tremendous

[1]https://www.nist.gov/cyberframework

potential, and have the opportunity to make considerable changes to areas of incident response.

Self-healing, and closely related terms like automated remediation, self-recovery, self-repairing, fault-tolerance and rapid recovery, all aim to enable a system to, without human interaction, remediate to a safe state autonomously after detecting malicious activity. This is closely related to cells' biological ability to repair themselves after being exposed to damage automatically, and have inspired the same phenomenon in computer science. Using the analogy of an ill patient, in order to restore a person to a healthy state, options like amputation and surgery act as permanent solutions, while sedatives and antidotes may be sufficient to keep the disease restrained.

As Ghosh et al. state in their survey of self-healing systems [GSRU07], increased system complexity will in return make rectification of system faults and destructive attacks more "*difficult, labor-intensive, expensive, and error-prone*". The idea of a self-healing system being able to autonomously restore a system and repair vulnerabilities would reduce these difficulties, since human interaction with the system is kept to a minimum. A practical example would be if a web server had an SQL injection[2] exploited, and the contents of the database were erased. Rather than having to remediate the system and manually look for the vulnerability, the self-healing system would perform these actions automatically as well as reconfiguration of input handling. Despite the simpleness of this example, there is interest in investigating if such line of thought can be applied to more intricate scenarios.

In 1984, the creation of the first packet inspection technology, named IDES, occurred [Bru01] and was the beginning of a new security mechanism targeting network traffic. Despite the fact that the development of intrusion detection systems (IDSs) and intrusion prevention systems (IPSs) since then have greatly improved and are essential components in security infrastructure, there is reason for concern to whether they will provide satisfying results in today's and the future's virtual threat landscape. Keromytis predicted in his paper published in 2007 [Ker07] that network-based reactive protection mechanisms were likely to be inadequate in the future. He anticipates so since firewalls tend to become congestion points due to the increasing use of computation-intensive protocols like IPsec, the complexity of protocols makes packet inspection more advanced, encryption of packets renders inspection useless, and malware use cloaking techniques like polymorphism to bypass the inspection technologies. It is therefore a demand for new, autonomous mechanisms to ensure security for these systems, when both detection and prevention procedures are not impenetrable.

[2]Injection of SQL statements which unknowingly run on the database. An attacker may make modifications to the database like deletion or creation of rows, retrieve data from the database, etc.

1.2 Specialization project

During the fall of 2019, approximately 12 weeks were spent preparing for this thesis in a specialization project. Preparatory work such as researching the field of self-healing, planning which tools and platforms to use, which methodology to follow and performing preliminary experiments were conducted. The goal of the specialization project is, in general, to prepare for the master thesis by narrowing down and specifying the problem tasks. From the specialization project, we were able to decide that web vulnerabilities in PHP would be the main problem area to focus on in the master thesis, in addition to achieving a greater understanding of self-healing systems. The research questions from the specialization project are given below:

RQ1 What are the potential benefits of using Docker containers in the goal of achieving self-healing systems rather than using non-virtual systems?

RQ2 What kind of self-healing techniques already exist, what are the shortcomings of these and how can one overcome these?

RQ3 Which vulnerabilities that enables attacks will the system be able to patch autonomously?

RQ4 Which vulnerabilities that the system were unable to autonomously patch, will the system be able to immunize?

RQ5 If vulnerabilities are able to heal, is there a correlation between the self-healing techniques?

However, the research questions have been modified throughout the project and are precisely stated in the following section.

1.3 Research questions

There has been conducted limited research in the field of self-healing with regards to automatic remediation of intentional cyberattacks, and to an even smaller extent on self-healing with regards to automatic healing of vulnerabilities after cyberincidents. In this thesis, investigation of existing self-healing research is performed, as well as developing a self-healing script to investigate certain, self-healing techniques with regards to cyberincidents. The research questions form the basis of the research area and shall be answered throughout the thesis. The following research questions will be addressed:

RQ1 What kind of self-healing techniques already exist, what are the shortcomings of these and how can one overcome these?

RQ2 Which vulnerabilities that enable attacks will the system be able to patch autonomously, and which ones will it be able to immunize autonomously?

RQ3 If vulnerabilities are able to heal or be immunized, is there a correlation between the self-healing techniques?

1.4 Outline

The structure of the master thesis is presented below:

- **Chapter 2:** Background and Related work. Presentation of relevant terminology, technologies and definitions, along with related work to the research topic.
- **Chapter 3:** Methodology. Introducing research methodology, including how and why tools and approaches were chosen and how they shall be used.
- **Chapter 4:** Experimental setup. Presents which tools and platforms are chosen, and how they are used in the technical experiments.
- **Chapter 5:** Experiments and Results. Demonstrates execution of technical experiments using the developed script, and their associated results.
- **Chapter 6:** Discussion. Gives a discussion of the results, taking the research questions into account and discusses future work on the topic.
- **Chapter 7:** Conclusion. Concludes and summarizes the thesis.

Chapter 2
Background and Related work

In this chapter, we present terminology related to self-healing, other topics related to the thesis and related work in the field. The chapter is structured as follows; the term self-healing is examined in-depth, as well as cybersecurity principles, the programming language PHP, and Docker containers are reviewed. Further, related work is presented, and the corresponding key findings from the literature review.

2.1 Self-healing

The term "self-healing" is a broad term, associated with a plethora of definitions and appliances. In the following section, we present definitions, concepts, and similar terminology of the topic, such as fault tolerance and autonomic computing.

2.1.1 Definition and Classification

Self-healing is defined in several ways, and we present some of the following definitions from other literature:

> *"**Self-healing** is an approach to detect improper operation of software applications and transactions, and then to initiate corrective action without disrupting users."* [RRS11]

> *"**Self-healing** can be defined as the property that enables a system to perceive that it is not operating correctly and, without (or with) human intervention, make the necessary adjustments to restore itself to normalcy."* [GSRU07]

> *"The term **self-healing** for software is inspired from the biological healing process for human and animals, where the body heals itself by repairing the*

> *affected tissue or bone, the process of healing is carried out internally from inside the body, the cells will gather in the place that has been affected (ex. Tissue insured or bold vessel cut) and the heal process retain the affected place to its original health status."* [HFAAF17]

A self-healing system can be described in various ways. With respect to computer systems, self-healing is the ability to remediate unintentional or malicious actions done to the system. From the given definitions and other related work, it becomes clear that use cases, implementations, concepts, research, and appliances of self-healing differs. Examples of different interpretations of self-healing procedures and concepts are presented below:

Example 1 The system's healing mechanism acts based on the assumption that unwanted behavior is a result of either bugs, attacks, or both.
Example 2 The system's healing mechanism is inspired by biological phenomenons in cells' regenerative properties.
Example 3 The system's healing mechanism assumes that healing of symptoms that cause unintentional actions, rather than healing the root cause of these actions, is sufficient.
Example 4 The system's healing mechanism heals by remediating the system to a safe state when unexpected changes happen in the system.

2.1.2 The concept of Self-healing

Self-healing is the ability of a system to be able to remediate malicious actions done to it. A crucial distinction which is of importance in this thesis is the contrast between self-healing systems, which *remediate or restore the system to a known, safe state,* and self-healing systems, which *remove the vulnerability* that allowed the attack to happen. Whereas the first type of self-healing contains much of the functionality of the latter type like self-monitoring and self-recovery, the second type aims to self-harden and self-repair itself.

The self-healing process, or life cycle, varies based on the system it is applied and the thresholds of the system. Ghosh et al. state that a "healthy/broken"-scale can be implemented in self-healing systems [GSRU07]. When the system reaches a broken state, the self-healing mechanism needs to regulate the system in such a way that it becomes healthy again. An example is resource allocation; if one server receives an overwhelming load of traffic and reaches a broken state, the self-healing system must allocate traffic to another server to restore the healthy state. With regards to this master thesis and using this scale in an attack scenario, the self-healing system would announce the system as broken if, for instance, there are findings of an attack like spyware on the server.

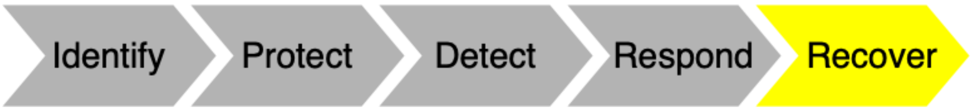


Figure 2.1: Five steps to ensure security, whereas self-healing takes part in the fifth step.

Following a modern approach to securing digital infrastructure, there are a few necessary steps which need to be fulfilled, illustrated in figure 2.1. According to NIST's Cybersecurity Framework[1], the first step is to identify threats by doing risk assessments and asset management. The second step ensures security by protection of data, which is classical data security like access control, implementing protection technologies, and awareness training. IDSs and IPSs often provide the detection of cyberattacks. If there has been a breach, the response often includes incident response teams and remediation of malignant modifications. Self-healing plays a part in the fifth and final part, which is to recover. Within this step, self-diagnosis must be made in order to evaluate what happened to the system and what vulnerabilities are present. The result from this phase is used to self-harden the system, to ensure that an equivalent exploit cannot be made. Whereas the first four steps are highly automated, the remediation step is still mostly manually managed.

2.1.3 Related terminology

The self-* properties of reliable computer systems

There is a plethora of self-* properties which are closely related to self-healing, yet not equivalent. Self-* systems refer to systems able to self-manage and suggests methods for developing highly-reliable, self-managed, complex computer systems [She08]. Table 2.1 summarizes Hudaib et al. definitions of eight self-* properties [HFAAF17].

The definitions are clearly not mutually exclusive, and become intertwined with each other. One could say that self-healing is composed of multiple of these definitions. In the self-healing system STING by Brumley et al. [BNS07], self-monitoring, self-diagnosis, self-hardening and self-recovery make up the main components of self-healing.

[1]https://www.nist.gov/cyberframework

Self-* Properties	**Definition**
Self-Adaption	The ability to enhance its current status and evaluate it.
Self-Optimisation	Finding the optimal solution to meet its goals.
Self-Monitoring	Being able to monitor its internal functions and performance.
Self-Testing	The ability to test oneself and evaluate if malfunctions are present.
Self-Diagnosis	Identification and diagnosis of oneself to reduce errors.
Self-Management	Managing its own functions without human intervention.
Self-Control	The process of controlling the state and behaviour of the agent.
Self-Configuration	A process by which components are configured by themselves or by a dedicated configuration component.

Table 2.1: Definitions of self-* properties.

Fault-tolerance

> *"A fault-tolerant system should be able to handle faults in individual hardware or software components, power failures, or other kinds of unexpected problems and still meet its specification."* [Dub13]

Fault-tolerance and self-healing have similarities in their definition, where both should be applied in systems that need to be robust and reliable. Fault-tolerance often does so by adding redundancy to the architecture to resist failures, while self-healing tries to repair the system when exposed to unintended behavior. However, self-healing aims to identify, mitigate, and, preferably, eliminate the root cause of unwanted actions, whereas fault-tolerant design principles' primary objective is remediation to a state where it can continue execution [Ker07]. Based on the difference in these characteristics, one could say that fault-tolerance is targeted towards rare and mostly unintentional failures, which are hard to mitigate, whereas self-healing should protect systems from intentional attacks from adversaries. Viewing the two terms from an autonomic computing perspective, self-healing is a modern approach to handling system failure [Per13].

Biology and Autonomic Computing

Organisms' have the ability to self-repair after being prone to illnesses and wounds. When someone breaks their leg or catches a cold, the human body automatically heals itself to the best of its ability. The human nervous system is considered the most intricate autonomic structure existing in today's nature [PH04]. The brain acts as the controller of a large network, which monitors changes and responds appropriately to events. This phenomenon has inspired computer systems to implement self-healing systems similar to the ones running in our bodies. The humane immune system

(HIS) can distinguish between its own benign molecules ("self") from malignant ones ("non-self"), which is an essential feature. Comparing an animal to a machine, an animal may act unconsciously to an event due to its autonomic nervous system. The machine, on the other hand, is currently not in possession of this ability.

Implementing similar biological abilities which the brain and nervous system possesses, is called Autonomic Computing Systems (ACS) [HFAAF17]. Like with the human body, the computer systems should have the properties self-configuration, self-optimization, self-healing, and self-protection [KC03]. A concrete example of such implementations are malware, like viruses, and remedies, like antivirus software.

Prevention systems

When speaking of vulnerabilities and susceptibilities towards computer systems, there exists several technologies and concepts trying to prevent, mitigate and detect malicious actions. These are similar to self-healing ideas and might be easy to mistake for one another, when they in reality use diverse techniques and approaches.

Intrusion Detection Systems (IDS) IDSs use methods in real-time to detect attempts or actual access to systems by unauthorized parties [Row02]. The IDS usually alerts the system administrator of the malicious actions which have taken or take place.

Intrusion Protection Systems (IPS) IPSs are identical to IDSs with the exception that these systems *prevents* the malignant operations by for instance blocking detected, harmful pieces of code [Lim06]. Both IDSs and IPSs are commonly used in protection against network-based, destructive actions.

Antivirus Software (AV Software) AV Software is a piece of software installed on a computer in order to give better protection than what the underlying operating system can provide. It is most commonly a preventive solution, but it may also try to remove infected programs or software [KB15].

2.2 Cybersecurity

Due to the increase in cyberattacks and the fear of cyberwarfare, the investment and research in cybersecurity has accelerated over the past few years. More than 50 nations have created and published strategic documents stating their official stance on cyberspace, cybercrime and cybersecurity [20112]. The *Merriam Webster* dictionary defines the cybersecurity as:

> *"Measures taken to protect a computer or computer system (as on the Internet) against unauthorized access or attack."*

Cybersecurity is a vital component of any digital system to be able to ensure the required non-functional requirements of a computer system. The National Institute of Technology: https://www.nist.gov (NIST)[2] defines cybersecurity as:

> *"Prevention of damage to, protection of, and restoration of computers, electronic communications systems, electronic communications services, wire communication, and electronic communication, including information contained therein, to ensure its availability, integrity, authentication, confidentiality, and nonrepudiation."*

Cybersecurity is often confused with information security. Whereas cybersecurity focuses on protecting computer systems from unauthorized access, information security is more comprehensive, including the protection of both digital and analog assets.

2.2.1 The basics of cyberattacks

Cyberattacks are intentional, malignant acts towards a cyberservice and stand out from unintentional bugs or misconfigurations, which may cause disruptions. NIST defines cyberattacks as:

> *"An attack, via cyberspace, targeting an enterprise's use of cyberspace for the purpose of disrupting, disabling, destroying, or maliciously controlling a computing environment/infrastructure; or destroying the integrity of the data or stealing controlled information."*

Following the methodology of Liu et al. [LC09], the concept of cyberattacks will be further delved into. An example of a cyberattack is illustrated in figure 2.2.

Why? With the existence of software bugs, configuration defects and design flaws, *vulnerabilities* become present. By taking advantage of this vulnerability, more accurately named *exploiting* the vulnerability, an adversary can *breach* the system and gain unauthorized access. The vulnerability can be remotely accessible, or might require *social engineering* performed by the attacker.

[2]https://www.nist.gov

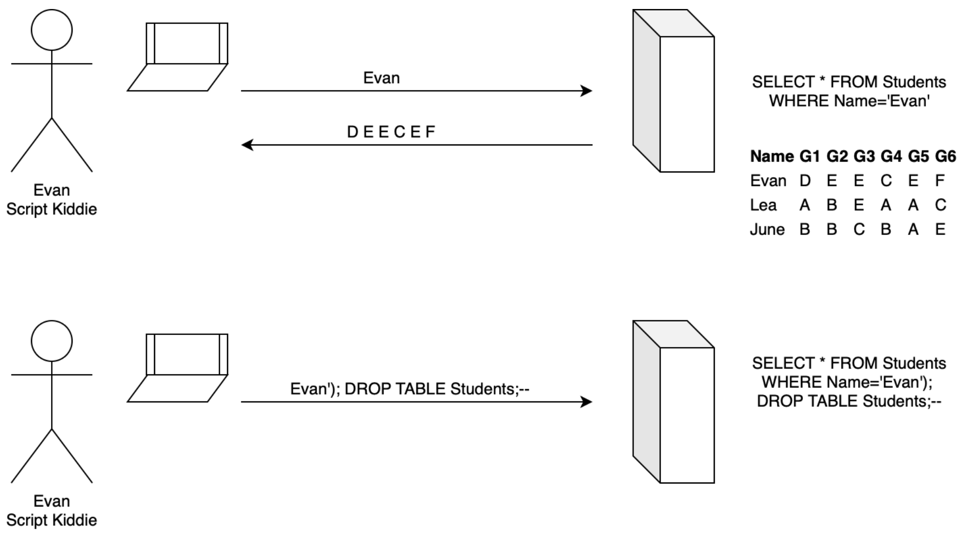

Evan
DEECEF
Evan
Script Kiddie
SELECT * FROM Students WHERE Name='Evan'
Name G1 G2 G3 G4 G5 G6
Evan D E E C E F
Lea A B E A A C
June B B C B A E
Evan'); DROP TABLE Students;--
Evan
Script Kiddie
SELECT * FROM Students WHERE Name='Evan'); DROP TABLE Students;--

Figure 2.2: Example scenario of who conducts cyberattacks, how it is performed and what is attacked; a script kiddie performs an SQL-injection towards the database which possesses the grades of school courses.

What? Cybersecurity challenges can be analyzed from three perspectives, being motive, means, and opportunity. Organized crime, hatred, terrorism and acts of war have become motivations for several malicious hackers. The internet makes the distribution of advanced cyberattack tools incredibly easy, as well as the attacks get increasingly sophisticated. Hence, cyberattacks become very easy to perform despite being hard to detect and defend. While the digitization escalates, the number of exploitable vulnerabilities grows as well.

Who? A cyberattacker gain access to computer systems with either benign or malicious intentions. The malicious attackers could be insiders, but the hacking often takes place remotely from the outside. Examples of cyberattackers are script kiddies, state actors, and hacker groups.

How? The adversaries usually utilize a range of hacking tools to execute attacks. In order to conduct a successful breach, hackers often do reconnaissance, scan targets, launch exploits on detected vulnerabilities, establish footholds by gaining access and make some sort of profit of the attack.

2.2.2 Principles for managing security

There are several guidelines one can follow and measures one can implement in order to secure a system. In *Grunnprinsipper for IKT-sikkerhet, versjon 1.1* [Nas18], Nasjonal Sikkerhetsmyndighet (NSM)[3] presents a set of principles to implement for establishing ICT security in an enterprise. The four main steps in the cycle of securing the computer systems are:

1. **Identification** and **mapping** of value chains, devices and software and users and their privileges.
2. **Protecting** emails, browsers, and other data in transit, secure configurations and design processes, establish suitable logging and control administrative privileges.
3. **Maintaining** and **detecting** malware and unauthorized changes by monitoring the system with IDS/IPS software, verifying configurations, creating backup solutions and performing penetration testing and red team exercises.
4. **Managing** and **remediating** cyberincidents by establishing strategies for incident response.

Managing security for any enterprise is a continuous cycle, and needs to be prioritized during all stages of development and production.

2.2.3 Common web vulnerabilities

As the internet, and services relying on this communication medium, become more prevalent in people's everyday lives, the more desirable the platform becomes for hackers. Since services provided on the internet are accessible online, the attackers can often execute attacks remotely. The Open Web Application Security Project (OWASP) Foundation[4] is an open-source organization with the goal of improving software security, having all resources publicly available. Periodically, the association publishes a report with the most common web vulnerabilities at the current time. *OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks* [OWA17] contains the top 10 greatest risks of web applications present in 2017. The vulnerabilities are summarized below:

Injection Untrusted data, often in the form of user input, is sent to a code interpreter without validating or sanitizing the input.

[3]https://www.nsm.stat.no
[4]https://owasp.org

Broken Authentication Enables manual or automatic mediums to gain access to a user account or admin account.

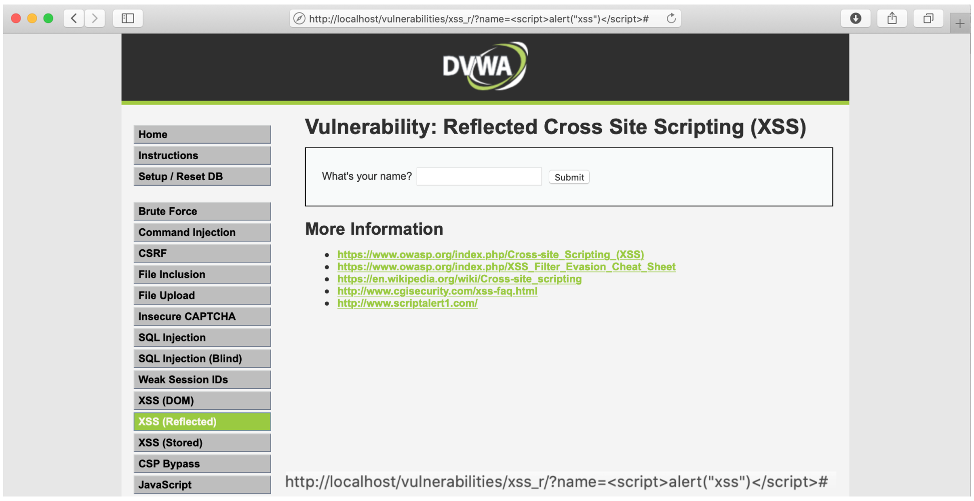
Sensitive data exposure Exposure of personally identifiable information (PII), for example, data breaches or unencrypted data in transit.

XML External Entities (XXE) A weakly configured parser processes an input containing a reference to an external entity.

Broken Access control Allows attackers to bypass authorization and often perform tasks as they were the administrator.

Security misconfigurations Using default configurations or displaying verbose error messages are common examples of misconfigurations.

Cross-Site Scripting (XSS) Enables malicious JavaScript code to run in a victim's browser. The vulnerability is exemplified in figure 2.3 and 2.4.


http://localhost/vulnerabilities/xss_r/?name=<script>alert("xss")</script>#

DVWA

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? Submit

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

http://localhost/vulnerabilities/xss_r/?name=<script>alert("xss")</script>#

Figure 2.3: Loading the page with specially crafted user input containing JavaScript in order to check whether the server contains an XSS vulnerability.

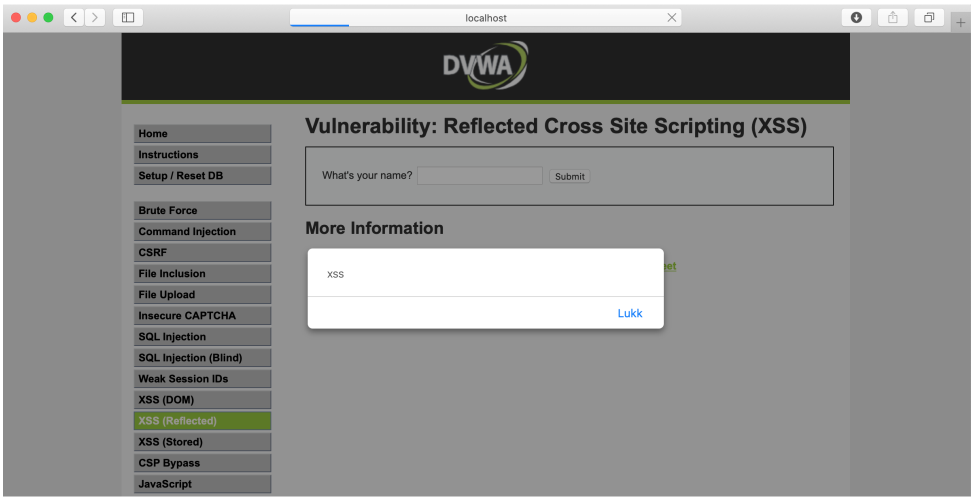


Figure 2.4: If the web page is vulnerable to JavaScript injections, the illustrated popup will appear when the page is run with the input from figure 2.3.

Insecure Deserialization Deserializing data from untrusted sources, which may result in DDoS attacks or remote code execution.

Using Components with Known Vulnerabilities Either using vulnerable components or not updating/patching vulnerable components.

Insufficient Logging and Monitoring It is recommended to implement logging and monitoring to detect attacks.

2.3 PHP

PHP (PHP Hypertext Preprocessor) is a free programming language created in 1994 for web development and runs on several operating systems, such as Linux, Windows, and Macintosh machines [Atk00]. The developers of PHP describe the technology as:

> *"a popular general-purpose scripting language that is especially suited to web development. Fast, flexible and pragmatic, PHP powers everything from your blog to the most popular websites in the world."*[5]

[5]https://www.php.net

Even though the programming language is more than 25 years old, it is still widely used. W3Techs declares that PHP is used by 78.3% of all websites in 2020[6], where the programming language of the server is known. From the servers which use PHP, 49.9% uses PHP version 5, and 49.7% uses PHP version 7. This makes PHP still highly relevant, even though there are several other technologies available.

Like other programming languages, PHP is prone to security issues. CVE Details[7] has registered 604 security vulnerabilities on PHP[8], whereas 53 on Node.js[9], 56 on Ruby on Rails[10] and 49 on Python[11].

2.4 Docker Containers

With regards to this master thesis, the isolation, simplicity and adaptability of Docker containers are the most compelling properties which will be utilized during the project.

2.4.1 What is a Docker container?

Docker containers was introduced in 2013 and are today a popular virtualization technology and microservice architecture introduced by Docker[12]. According to the company, there are more than 105 billion container downloads and at least 750 Docker Enterprise Customers. Docker defines containers and images as:

> *"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings."*

The containers may appear similar to virtual machines, but they utilize the kernel of the host's operating system differently. Whereas virtual machines construct an entire virtual operating system, multiple containers exploit the same kernel of the

[6] https://w3techs.com/technologies/details/pl-php
[7] https://www.cvedetails.com
[8] https://www.cvedetails.com/vulnerability-list/vendor_id-74/product_id-128/PHP-PHP.html
[9] https://www.cvedetails.com/vulnerability-list/vendor_id-12113/Nodejs.html
[10] https://www.cvedetails.com/vulnerability-list/vendor_id-12043/product_id-22568/Rubyonrails-Ruby-On-Rails.html
[11] https://www.cvedetails.com/vulnerability-list/vendor_id-10210/product_id-18230/Python-Python.html
[12] https://www.docker.com/company

host's operating system. Since the applications share the kernel, the size of the application is greatly reduced in combination with increased performance, illustrated in figure 2.5.

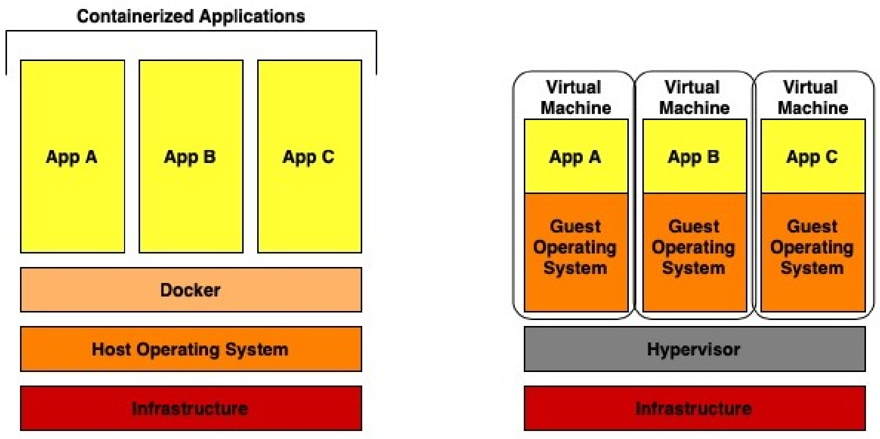


Figure 2.5: Illustration of Docker containers and virtual machines.

Containers achieve isolation by using namespaces and assigning a network stack to each container. The container is unable to see and interact with processes of other containers, including no access to sockets and interfaces of other containers, unless explicitly configured to do so. Also, with regards to isolation, control groups equally distribute resources like memory and CPU to each container. Control groups also prevents denial-of-service attacks such that one container cannot exhaust the available resources and take down the whole system.

Docker is a heavily loaded term often associated with multiple meanings. *Docker* uses a client-server architecture [JNS16]. Through the *Docker Client*, the user can run a set of *Docker commands* through the *Docker daemon* to interact with the *Docker host*. The daemon runs, builds and distributes *Docker containers*, in addition to publishing *Docker images* to *Docker registries*. Docker containers are spin up from docker images. An image is built locally, or pre-made images are downloaded using instructions stored in a special file. The *Dockerfile* is triggered upon a build request, where Docker reads the files for its instructions and returns the image. Docker registries make distribution of images easy, and the Docker public registry is named *Docker Hub*. The Docker container is an isolated environment which includes all elements necessary for an application to run.

2.4.2 Immutability

To be able to make changes to a running Docker container or the Docker image, the container needs to be brought down, the image of the container is modified, and then the new version of the container is spun up. One could set up the container with SSH[13] to patch it, but building new images each time there is a need for an update of source code or configurations could be a better idea. Often, multiple instances of an image is run. When the containers require modifications, the original image is rebuilt, and all new instances of the image are updated. Following this methodology, one cannot apply authorized changes to a Docker container. This gives Docker containers the property of immutability [MGK19]. The *Merriam Webster* dictionary defines "immutable" as *"not capable or susceptible to change"*, which agrees with the properties of Docker containers.

Briefly, such immutability has the potential to make intrusion detection and incident response easier. If there has been made modifications to the container, there ought to have been a breach - the design approach of Docker containers does not allow modifications to happen to a running instance.

2.5 Related work

In October 2001, IBM released a manifesto expressing that the greatest barrier in IT industry advancement, is the software complexity crisis. Jeffrey O. Kephart and David M. Chess summarizes the manifesto in *The Vision of Autonomic Computing* [KC03]. It is stated that as systems become more diverse and entangled, system developers will not be able to predict and design all actions among system parts. Hence, upcoming issues are to be dealt with at runtime. The emergence of the term "autonomic computing" occurs, which they briefly define as *"... computing systems that can manage themselves given high-level objectives from administrators"*, and state that the term is deliberately chosen with biological connotation. It is further examined what type of autonomic system might work in the future, and self-management, comprising of self-configuration, self-optimization, self-healing, and self-protection, are stated as elementary aspects of how autonomic computing will be in the future. The self-healing feature in autonomic computing shall *"... detect, diagnose, and repair localized problems resulting from bugs or failures in software and hardware"*, while the self-protection feature shall *"... defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. They also will anticipate problems based on early reports from sensors and take steps to avoid or mitigate them"*. For the purpose of this master thesis, IBM clearly separates the two aspects -

[13]The SSH (Secure Shell) protocol allows for secure system administration and file transmission over insecure networks.

self-protection and self-healing mechanisms as two separate research areas and, as of today, require separate approaches. It is also necessary to be able to create a fully autonomous system. As stated in the paper, autonomic computing offers multitudes of benefits, but nevertheless significant engineering challenges. This is perhaps why the field is still of substantial interest.

There has been conducted some research in self-healing with inspiration from biological processes. Elsadig et al. present a model for both intrusion prevention and self-healing for network security. [EA09] integrates an artificial immune system (AIS) with intrusion prevention inspired by danger theory and adaptive immune systems within immunology theory. The IPS triggers the self-healing mechanism in the event of malicious events or attack profiles. The self-healing mechanism, named self-healing agent (SHA), is an expert knowledge base trained to adapt to abnormal activities inspired by cell regeneration. It does so by generating "fix candidates" for each fault and repairs the damages. Afterward, it performs self-testing for the new component before it deploys it. The paper contributes to research within the field of biologically inspired intrusion prevention and self-healing systems. It does not involve a concrete example of the mechanisms in practice. With regards to self-healing and this master thesis, what is described has similarities to what is aimed to be done. However, the proposed model is hard to grasp practically since there is no reference to the system in deployment.

Joseph et al. propose an autonomic prediction model for automatic recovery of attacked virtual machines in cloud [JM19]. Their prediction model shall secure and recover the virtual machines by using a self-healing algorithm. The virtual machines remediate to safe, stable snapshots to self-heal. The predictive algorithm predicts which virtual machine and which of its snapshots are infected. From these results, it will automatically restore itself using a saved snapshot and therefore remove all infected parts. The result of the research showed that Support Vector Machine (SVM) algorithm outperforms Gaussian Naïve Bayes and Decision tree algorithm with an accuracy rate of 98.4% in determining the attacked virtual machine snapshots. Using self-healing in a virtual environment with virtual machines to prevent adversaries has similarities to what is explored during this master thesis. However, the self-healing technique suggested in this paper does not patch or remove the vulnerability present, like what this master thesis intends to do using Docker containers.

Redundancy in code, either it is due to the programmer's coding style or present unintentionally in the libraries or packages implemented, is often associated with reduction in performance and is not a desirable characteristic in code. However, fault tolerance techniques often incorporate redundancy to be able to withstand unfortunate events and increase reliability. Nicolò Perino proposes an interesting approach combining both redundancy in code and fault tolerance in order to self-heal software systems [Per13]. By going back in the execution and replacing the failing operations with redundant operations, allows the system to recover from failure. Three key elements are introduced, namely a state handling mechanism, a roll-back strategy, and a workaround selection criterion. A captivating property of this approach is that the healing strategy mitigates the failures by eliminating symptoms of the failure's origin, rather than eliminating the actual root cause. With that said, this approach is mainly targeted towards healing of unintentional software bugs, whereas this master thesis will focus on intentional attacks. Nevertheless, this is an innovative contribution to the field of self-healing, and it would be interesting to investigate further if the concepts of fault-tolerance, redundancy, and healing based on symptoms could be applied in self-healing systems targeted towards malicious attackers.

As a result of vulnerable software, attacks over the web are increasing, and this is mostly due to the lack of knowledge of software architects, developers, and designers. Jaffar et al. use public vulnerabilities data to self-heal these vulnerabilities in software systems automatically without human intervention [Jav]. Using input from Common Weakness Enumeration (CWE), the framework provides suggestions and auto-correction to remove present vulnerabilities in the code. Static analysis tools check the source code for potential vulnerabilities towards attacks, while the code transformation module applies necessary changes after scanning the source code. Referring to their prototype example, they achieved a code accuracy of 83% with a 6% syntax error. Scanning the source code for vulnerabilities in advance of production, is a valuable approach to removing bugs and security weaknesses. As to this master thesis focusing on the incident response part of security, it could be possible to implement similar scanners using public vulnerability data after security incidents have been identified.

Another contribution to the field of autonomic patching is the system "ClearView" by Perkins et al., described in [PKL$^+$09]. The system is able to automatically patch stripped Windows x86 binaries without any other information about the software, and without human interaction. The correction of errors and vulnerabilities are based on the expected behavior of learned invariants, and the architecture is composed of five main components. The model learns expected invariants by observing normal executions using Daikon. Monitoring or observing executions and determining whether they are normal or failed, is in their implementation done using Heap Guard and Determina Memory Firewall. The system must also find a set of correlated invariants that characterize normal and failed executions whenever the monitors detect failures. ClearView generates a set of candidate repair patches that enforce the invariant for each correlated invariant. It then evaluates and ranks the patches as they execute. The process of learning improves the quality of the patches over time, and the authors compare this process to a biological immune system. ClearView was tested in a Red Team exercise, where the results showed that it automatically generated corrective patches for seven out of ten exploits. The developers of ClearView make an interesting approach to self-healing with the use of learned invariants. However, the system is limited by the fact that it only affects Windows x86 binaries. Nevertheless, trying to apply similar strategies using learned data about normal and failed executions to patch software could be extended to this master thesis.

2.6 Literature review findings

In this section, we present the key findings from the literature review. The research papers we targeted, are mainly practical and implemented systems.

2.6.1 The research status of self-healing software systems

As predicted by IBM in 2001, the road to completely autonomous systems will be extremely challenging and will require innovative solutions by researchers and developers. Since the release of their manifesto, the interest in self-healing systems has dramatically increased. Self-healing systems have the ability to reduce operational costs through less human intervention, and the time spent to detect, find and repair damages will decrease. However, creating such systems have proven to be significantly demanding. There exist examples of systems that behave autonomously, hence possesses the ability to self-configure, self-protect, self-heal, and self-optimize. Still, while conducting this literature review, there have not been made findings of any such systems deployed in a large scale today. The bottleneck in the development of these architectures is frequently the self-healing aspect.

Multiple factors are contributing to the hardness of creating self-healing systems. When designing such architectures, it becomes evident that the systems need to possess abilities of self-awareness. What has happened? Where is the breach? What caused the breach? As one breach may have several root causes, it is difficult and comprehensive to pinpoint where the vulnerability is located. Even for a trained computer professional, finding and removing such susceptibilities is demanding. Transferring these abilities to a system, and have it act as an autonomous being, have proven to be greatly complex. Several solutions have been proposed in the literature reviewed, including approaches using machine learning, knowledge base consulting, exploitation of code redundancy, and systems inspired by biological approaches.

As summarized previously in the chapter, there are many self-* properties and other terms being used when approaching topics such as self-healing and autonomous systems. The ones that appear the most frequently, when speaking of self-healing systems, are concepts such as self-recovery, self-protection, automated remediation, and fault-tolerance. The words are used interchangeably, and the corresponding definitions of the term self-healing differ from paper to paper.

2.6.2 Classification of self-healing literature

Developing self-healing, or pseudo self-healing, systems can be approached in numerous ways, as described in the works in section 2.5. Even though there are significant differences in their procedures, it is possible to extract a few common characteristics from some of the literature. In this subsection, these key features will be highlighted, and an attempt to classify the researched literature will be presented. As one of the findings in this review, [HFAAF17] and [GSRU07] present some classifications and summaries of existing literature on self-healing.

Papers	**Healing approach**	**Incidents**	**Healing technique**
Elsadig et al. (2009)	Remove susceptibilities	Intentional	An expert knowledge base it trained to adopt to abnormal activities, inspired by cell regeneration mechanism, and generates fix candidates to repair specific damages.
Joseph et al. (2019)	Fault tolerance	Intentional	Identifies malicious snapshots of virtual machines using different machine learning algorithms, and reverts to a safe snapshot whenever a snapshot is predicted to be infected.
Perino (2013)	Remove susceptibilities	Unintentional	Combines self-healing and fault tolerance techniques by using code redundancy operations to workaround failing operations.
Javed et al. (2019)	Remove susceptibilities	Intentional	Removes vulnerabilities in code by applying articles from Common Weakness Enumeration (CWE).
Perkins et al. (2009)	Remove susceptibilities	Both	Observers normal execution to learn invariants in which describe normal behavior, and whenever a failure is present, the flow of control is changed until the invariant is true.

Table 2.2: Presentation and comparison of the main characteristics of the related work.

Main characteristics of self-healing literature

From figure 2.6, important features of covered literature are presented. First, the research papers are separated based on their kind of research literature. For the interest of the project, researching the functionality of existing systems was more appealing than theoretic concepts, as reflected in the figure. The subsequent differentiation is based on the stated definition of and approach to self-healing in the researched papers. The third and final main divider is whether the specific systems are intended for intentional attacks, unintentional faults, or both. A summary of the targeted papers is provided in table 2.2.

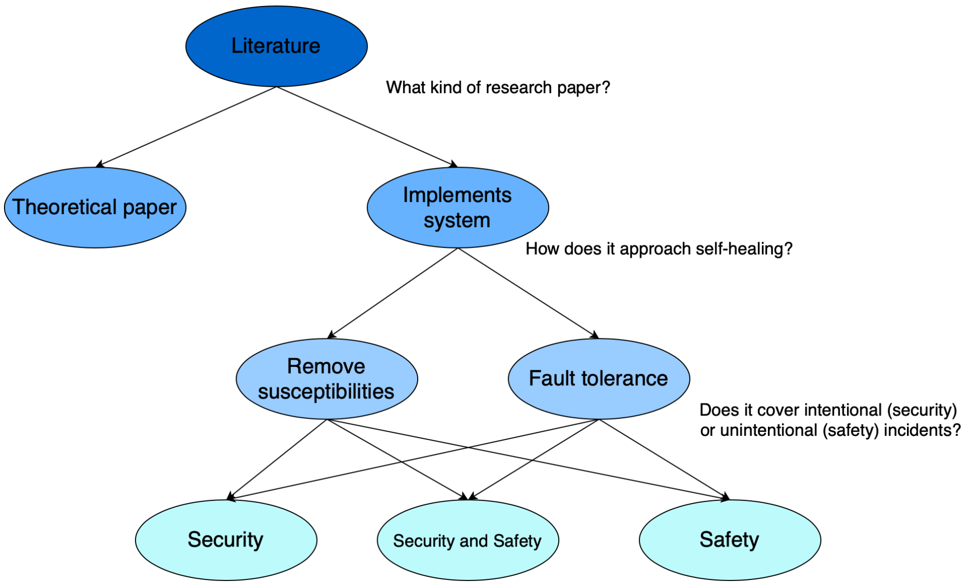

Literature
What kind of research paper?
Theoretical paper
Implements system
How does it approach self-healing?
Remove susceptibilities
Fault tolerance
Does it cover intentional (security) or unintentional (safety) incidents?
Security
Security and Safety
Safety

Figure 2.6: Key features used to categorize researched literature from the literature review.

Theoretical concepts/Specific systems The paper either covers the theoretic aspects of self-healing systems, for instance definitions, critical components, background and the life cycle procedure, or examples of developed and implemented self-healing systems. For the interest of the project, researching the functionality of existing systems was more appealing than theoretic concepts, as reflected in figure 2.6. The following listed papers are examples of specific self-healing systems being investigated:

- *"Biological inspired intrusion prevention and self-healing system for network security based on danger theory"* by Elsadig et al.
- *"Securing and self recovery of virtual machines in cloud with an autonomic approach using snapshots"* by Joseph et al.
- *"A framework for self-healing software system"* by Perino.
- *"Using public vulnerabilities data to self-heal security issues in software systems"* by Javed et al.
- *"Automatically patching errors in deployed software"* by Perkins et al.

Healing systems/Fault tolerance The systems presented all claim to be self-healing, but it becomes apparent that the definitions of the topic varies. As for this project, self-healing refers to the characteristic where a system is able to detect a vulnerability in which caused a security incident and have it autonomously remove, patch, or make the susceptibility unreachable for adversaries. Fault-tolerance, referred to as self-healing in at least one of the papers (for instance [JM19]), on the other hand, only brings the system to a condition from which it can proceed execution. Keromytos states that fault-tolerant systems *"can be viewed as primarily geared against rarely occurring failures"* [Ker07], but this is not always being practised.

Security/Safety The majority of the researched systems cover either intentional cyberattacks (security) or unintentional faults (safety), such as software or hardware bugs leading to system failures. One of the papers at hand ([PKL^{+}09]), claims that their system can handle both intentional and unintentional, disturbing incidents.

Self-healing techniques The five example systems listed previously, all use distinct techniques in order to achieve self-healing capabilities.

Chapter 3 Methodology

In order to conduct research in an efficient, sound and qualified manner, the approach, measures and techniques used should be well documented and follow a purposeful process. Kothari defines research methodology as [Kot04]:

> *"Research methodology is a way to systematically solve the research problem."*

The methodology often consists of several research methods to answer the problems in question. For this thesis, literature review is an example of a research method used in the methodology. Kothari differentiates the two [Kot04]:

> *"... when we talk of research methodology we not only talk of the research methods but also consider the logic behind the methods we use in the context of our research study and explain why we are using a particular method or technique and why we are not using others so that research results are capable of being evaluated either by the researcher himself or by others."*

In this chapter, we present the research methodology and methods used in the thesis. This includes literature review to research the current status of self-healing, selection of tools and platforms to use in the practical approach to self-healing, presentation of healing techniques for web vulnerabilities, and implementation and measurement of self-healing in the developed script.

3.1 Literature review

Literature review is an objective, critical analysis of published literature on a certain topic. By summarizing multiple studies in the field, inadequate research in the problem area becomes present. For this project, literature review serves multiple purposes. It is essential in answering RQ1, but also lays a foundation for the remaining research questions. According to Chris Hart in *Doing a literature review: Releasing the research imagination* [Har18], the definition of a literature review is:

> *The selection of available documents (both published and unpublished) on the topic, which contain information, ideas, data and evidence written from a particular standpoint to fulfil certain aims or express certain views on the nature of the topic and how it is to be investigated, and the effective evaluation of these documents in relation to the research being proposed.*

The motivation for conducting a literature review when researching a topic is diverse. Iivari et al. state that an essential first step for any research project is knowing current status of the body of knowledge (BoK)[1] in the research field [IHK04]. Levy and Ellis give five motivations and means for achieving this step through a proper literature review [LE06]:

1 The researcher becomes informed of the current BoK, hence where excess research exists and where new research is needed.

2 Provides a theoretical foundation for the research area.

3 Confirms the presence of the research problem.

4 Justifies the proposed research as a contribution to the BoK.

5 Shapes valid research methodologies, approaches, goals and research questions for the proposed study.

The literature review carried out in this project follows the principles of Cronin et al. [CRC08], and is further described detailed.

3.1.1 Selecting a review topic

When selecting the topic to be reviewed, it is wise to refine the topic to such an extent that the final amount of literature and papers produced is manageable. If the volume of information gets too large, the review will be either too long or too superficial. It might be tempting to choose broad search words to cover a great load of material related to the review topic, nevertheless it is not advisable.

[1]Body of knowledge: The set of concepts, terms and activities comprising a domain or field of profession

Despite the research questions not being carved in stone at the beginning of the project, the review topic was decided early in the process. Relevant topics for the assignment are *"self-healing computer systems"*, *"self-recovery computer systems"*, *"self-repairing computer systems"*, *"auto remedation computer systems"*, and other variations of self-* properties (2.1.3).

3.1.2 Searching the literature

After having narrowed down the review topic, it is necessary to examine the topic in a structured way to acquire satisfying information. Literature searches have become easier to conduct due to electronic databases containing heaps of articles. In this literature study, the search engines Google Scholar[2], IEEE Xplore[3], Microsoft Academic[4], Semantic Scholar[5] and CORE[6] are used.

When searching, Boolean operators like "AND", "NOT" and "OR" explicitly decide which search words to be included or not in the information generated. A relevant search phrase for this project is `("self-healing" OR "self healing") AND ("computer system" OR "software")`.

The time frame from which the search is performed is also important; for some review topics, elderly information might be obsolete. For the research area examined in this literature, the research is scarce, and therefore the papers included ranges from publications from 2003 to 2020.

When undertaking a literature search, deciding if publications should be included in the literature review can be based on the type of source. Table 3.1 summarizes the four kinds of sources. For this review, mainly primary source, secondary source, and conceptual/theoretical sources have been used.

Source	**Definition**
Primary source	A report by the original researchers of the paper.
Secondary source	Ex. a review article, summary of the paper written by someone else than the original researcher.
Conceptual/theoretical	Description or analysis of theories or concepts associated with the topic.
Anecdotal/opinion/clinical	Views and opinions about the field that are not research, reviews or theoretical. Clinical embodies case studies or reports from clinical settings.

Table 3.1: Sources of publications and definitions.

[2]https://scholar.google.no
[3]https://ieeexplore.ieee.org
[4]https://academic.microsoft.com
[5]https://www.semanticscholar.org
[6]https://core.ac.uk

As with all research, searching for relevant literature must be limited with regard to the time aspect. For some tasks, investing lots of time in finding relevant papers is crucial to the result of the project, whereas for other tasks, it is not necessary. In this project, there has been devoted a considerable amount of time to research the problem field.

3.1.3 Gathering, reading and analysing the literature

There are useful strategies for analysis and gathering of information that will help with the writing of the review. At first, the number of articles might be overwhelming; therefore, it is advisable to start categorizing them after reading their abstract or summary. When the papers have been sorted by type, it is necessary to read them more systematically and critically. Typically, one is concerned with the title, author, motivation, methodology, findings, and outcomes of the research papers. It is also recommended to read the articles, with the problem question(s) in mind.

3.1.4 Writing the review

When writing the review, the findings must be presented objectively in a clear and consistent way. The literature review of this project is represented in Background and Related work (chapter 2). The most interesting, diverse, and relevant articles found during the search have been more thoroughly described in section 2.5.

3.1.5 References

References to the papers in the literature review is an essential part of the process. For this project, all references are located at the end of the report in the References chapter.

3.2 Selection of tools and platforms

In order to carry out the project, choosing appropriate tools and platforms is necessary. First of all, a platform in which contains vulnerabilities must be chosen. Such a testbed must be compatible with one or more vulnerability scanners. The vulnerability scanner is used to discover vulnerabilities in which may have caused a security breach. Such scanners will be selected carefully based on the stated evaluation criteria. Since the healing should preferably be performed autonomously, the tools chosen must possess functionality in which can be automated. When selecting suitable tools and platforms for the thesis, the research and evaluation of such assets are critical concerning the achievement of desirable results. Section 4.1 describes the results from the selection in depth.

3.2.1 Test server

The platform in which hosts vulnerabilities to be exploited and healed is the testbed of the project. The testbed acts as an isolated environment where vulnerabilities are present, attacks are launched, and the experimentation of different self-healing techniques takes place. Since this project targets healing of web vulnerabilities using PHP, having a vulnerable web server based on PHP as the testbed becomes a reasonable choice.

Generating the testbed

When creating a web server as the testbed, the researcher may either develop a suitable server for the experiments by themselves or explore whether there exists open-source code of servers possessing characteristics such as vulnerabilities caused by faulty PHP use. The former option has the researcher creating, designing, and coding the web server from scratch, while the latter option has the researcher implement a pre-made construction of a web server.

Evaluation criteria

When designing and choosing the testbed of the project, there are several desirable traits in which the web server should have. In order to evaluate the testbed candidate, a few criteria must be considered:

1 Does the platform cover a variety of web vulnerabilities in which can be exploited?

2 Does the platform cover web vulnerabilities of multiple complexities?

3 Is the platform, or can the platform be, written in PHP?

4 Is the platform easy to manage, make modifications to, and integrate with other tools?

5 Can the platform be implemented within a reasonable time frame?

There is no clear weighting between the criteria, but the conditions concerning PHP as a programming language and the time frame will be considered the most important criteria.

3.2.2 Vulnerability scanners

After having exposed the test server to a security incident, a static vulnerability scanner is needed to reveal susceptibilities in the code. These revelations will further

Tool	Description
graudit[8]	Applies script and signature sets using the GNU utility `grep` to find security flaws in source code.
phpcs-security-audit[9]	Is a set of PHP_CodeSniffer rules that finds security vulnerabilities in source code.
progpilot[10]	A static analyzer of source code based on a constructed control flow graph.
RIPS[11]	Tokenizes and parses source code files into a model to detect vulnerabilities during program flow.
SonarQube[12] with SonarScanner[13]	Offers 189 rules for static source code analysis.
VisualCodeGrepper[14]	An automated code security review tool which identifies erroneous and insecure code.

Table 3.2: Brief summary of relevant scanners.

be used to heal the server. The scanner is necessary in order to automate the process of patching vulnerabilities in the PHP code.

Use of scanners

Vulnerability scanners are automated tools in which scans the given source code and looks for coding practices which may lead to security vulnerabilities and bugs. They are often useful because of their ability to scale well, can be run multiple times and gives precise, often verbose, feedback. However, the coverage and results from most scanners are limited. Intricate susceptibilities such as access control problems could be severe vulnerabilities, but are too complex for a scanner to detect. The scanner will also provide false positives and false negatives, which reduce the overall efficiency and performance of the self-healing system.

Available scanners

When researching available scanners for source code analysis, OWASP's list of source code analysis tools[7] was used. The list includes a plethora of analysis tools ranging from open-source to commercial tools, and some of these will be explored further. Nonetheless, there exists scanners which are not covered in this list and will not be covered in this project due to time constraints. Examples of relevant PHP-compatible scanners are briefly presented in table 3.2.

Evaluation criteria

The scanner is one of the main components in the self-healing process, and must therefore be selected cautiously. When exploring the assortment of scanners available,

[7]https://owasp.org/www-community/Source_Code_Analysis_Tools
[8]https://github.com/wireghoul/graudit/
[9]https://github.com/FloeDesignTechnologies/phpcs-security-audit
[10]https://github.com/designsecurity/progpilot
[11]http://rips-scanner.sourceforge.net
[12]http://www.sonarqube.org
[13]https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/
[14]http://sourceforge.net/projects/visualcodegrepp/

these must be evaluated based on stated criteria. For this project, the features coverage, integration and management, cost, and PHP support are of importance in order to achieve desirable results and remain within scope. Each criteria is equally weighted (25%), hence each feature is equally essential for the selection of a scanner.

Coverage The scanner must be able to discover a range of vulnerabilities to fulfill criteria with regards to coverage. It should be able to detect less complex vulnerabilities such as SQL injections, stored and reflected XSS, and remote executions using the function `shell_exec()`. More sophisticated security matters, such as authentication issues, cannot be expected to be detected by the scanner.

Integration and management As a means to achieve an autonomous system through the aspect of self-healing, the scanner must be integratable with the chosen testbed and other software. Therefore, it is also required that the tool can be run from the command line, without having to manage a graphical user interface, to automate the system. It should preferably be easy to set up and use, as well as being regularly maintained and updated.

Cost/availability It is a necessity that the tool is either open-source, free to use, or has a trial subscription with unlimited or a large number of scan runs.

PHP support Since the thesis investigates web vulnerabilities caused by unfortunate PHP scripting, the scanner must support static source code analysis of PHP files.

With regards to performance metrics such as false positive rates, false negative rates, detection rates, accuracy rates, and precision rates, these are difficult to measure in this thesis since the sample of vulnerabilities is quite small, and several of the vulnerabilities present are too complex to be detected. Therefore, these metrics are not considered explicitly in any feature.

3.3 Healing technique for web vulnerabilities

Self-healing systems are hard to comprehend, and therefore also to design. Such problems may be approached by *dividing and conquering*, that is, dividing a composite problem into two or more smaller, less hard problems. By solving these subproblems individually and combining each subsolution, a solution to the original problem is produced. For the project, five subproblems, or steps, are addressed in order to achieve self-healing characteristics on the testbed. Each step is a healing mechanism, having various consequences for the testbed based on the technique and means of healing. Following the principle of dividing and conquering, each subproblem should be approached and solved. However, unlike this algorithm, the fifth step is the ideal solution to the original problem, and the previous steps are iterated through to evaluate whether the fifth step is reachable for the given vulnerability. The five healing approaches do not rely on or build on top of each other. In order to evaluate how well each step and healing mechanism performs, evaluation criteria must be established and considered for each step.

3.3.1 5 steps

The practical approach of self-healing has been divided into steps, on the basis of making the problem more understandable and approachable. When a breach is detected, the healing will take form as one of the five steps. The last four presented steps will make use of the vulnerability scanner to identify the vulnerability. Following, we present the five proposed healing mechanisms.

Step 1 - Power off web server

Since the definition of self-healing involves removing the vulnerability in which caused the breach, powering off the machine or taking down the web server would prevent the vulnerability from occurring again. Since the service becomes unavailable, it is impossible to launch any attack successfully towards the target.

Step 2 - Remove susceptible PHP file

If it is possible to locate the PHP file in which contains vulnerable code, removing this file would stop future, equivalent attacks as the vulnerability being exploited is no longer present as the susceptible file have been removed.

Step 3 - Remove susceptible line of code in PHP file

The step resembles the previous one, whereas instead of deleting the susceptible PHP file, the line of code containing vulnerable code being exploited in the security breach is removed.

Step 4 - Add sanitization to input

Since the vulnerabilities often are triggered by specially, crafted user input, the step applies general sanitization to all user input. It tries to use distinct filters based on the indication of the detected vulnerability to customize the healing better.

Step 5 - Correct susceptible code causing vulnerability

The ideal step is where the vulnerability is completely healed in the PHP file. The necessary modifications to the PHP code are made to ensure that the exploit launched no longer is applicable to the server.

3.3.2 Evaluation criteria of healing techniques

In order to decide how suitable a healing mechanism is, evaluation criteria in which each technique will be measured against must be established. The healing evaluation criteria shall describe how well each step does when executed. The criteria include both functional and non-functional requirements, which are commonly used to describe vital quality characteristics and functionality of software systems [CN95]. For instance, although availability of the server is greatly reduced, turning off the machine is highly effective with respect to healing. This approach satisfies the healing criteria, but not the availability criteria. For the project, self-healing, healing or automatic patching, is considered a functional requirement. Further definitions and requirements for the evaluation criteria will be established in the next subsections.

Functional requirements

Functional requirements are functions *"that a system (...) must be able to perform"* [15990], hence specific functionality that is critical for the system to work as intended. They describe tasks and input/output behavior of the system. To have the self-healing approaches described in subsection 3.3.1 be evaluated, the following functional requirement is considered:

1 Healing: The self-healing procedure must remove the vulnerability or susceptibility at hand in the testbed, and make it unreachable and/or unexploitable.

Non-functional requirements

Non-functional requirements are *"used to delineate requirements focusing on "how good" software does something as opposed to the functional requirements, which focus on "what" the software does."* [PK04]. Therefore, the requirements describe properties, qualities and behaviors of the system, rather than specific tasks and functionality.

One can separate between basic and extra quality non-functional requirements [Fre87]. Basic non-functional requirements include the most fundamental qualities, such as functionality, reliability and safety, whereas extra non-functional requirements includes flexibility, documentation and enhanceability.

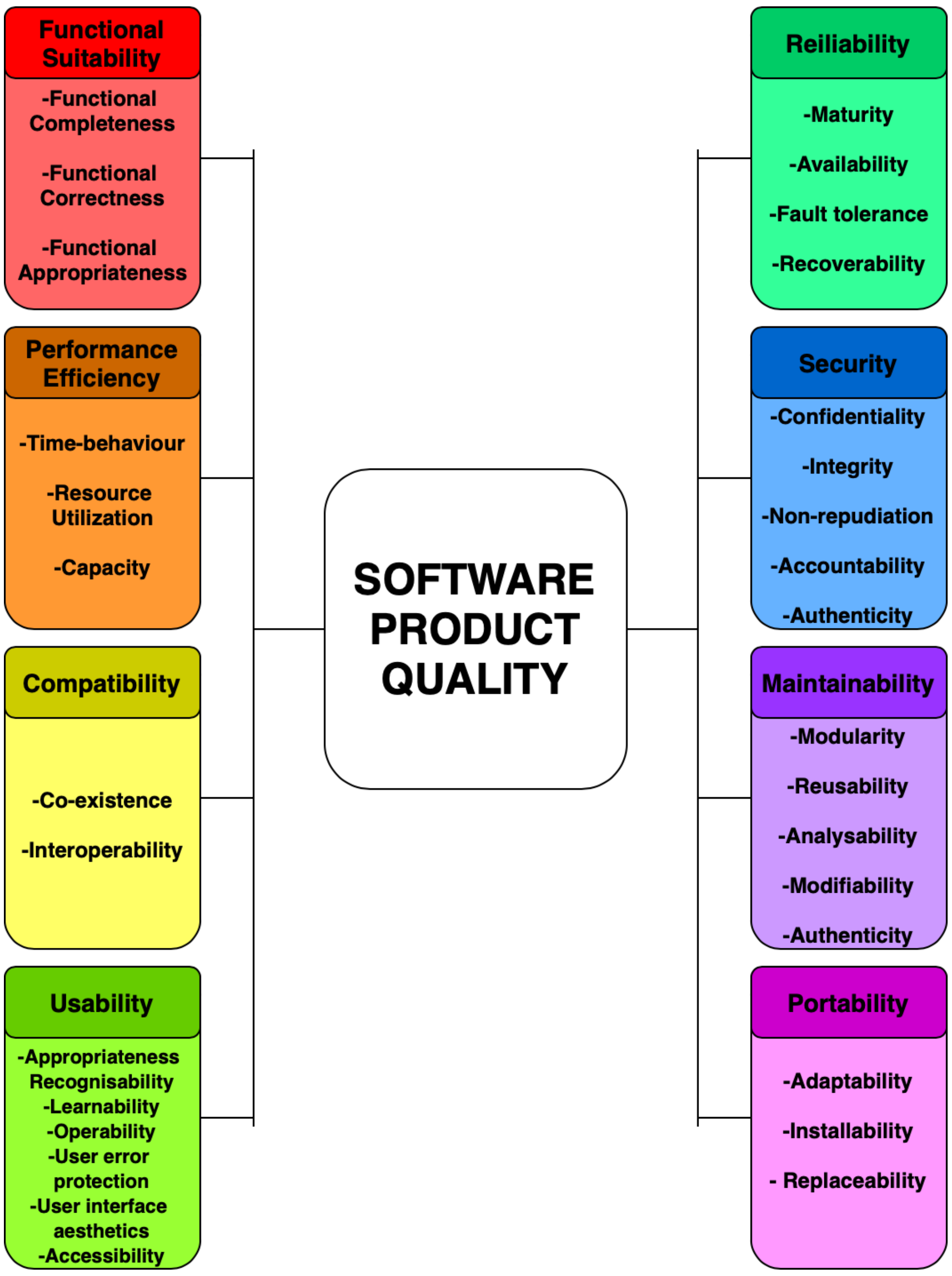


Figure 3.1: Illustration of characteristics and associated subcharacteristics from ISO/IEC 25010:2011.

The NFR Framework [MCN92] introduces the concept of softgoals for non-functional requirements and hardgoals for functionl requirements; softgoals are being pursued in a "good-enough" sense, while hardgoals need to be addressed absolutely. The NFR Framework defines the notion of "satisficing" to describe that a softgoal is satisfied in the good-enough approach. The framework assigns a criticality value to each softgoal as well.

As mentioned by Martin Glinz in [Gli07], several works classify non-functional requirements into subcategories. The ISO/IEC 25010:2011 proposes a product quality model with eight characteristics and associated subcharacteristics, which can be applied to both software and computer systems [20113]. The product quality is categorized into these characteristics. The non-functional requirements are illustrated in figure 3.1.

When choosing which non-functional requirements to incorporate and base the system development on, it is often desirable to incorporate all main characteristics, such as performance, reliability, usability, and safety. For the project, only a few selected, non-functional requirements are included in the evaluation criteria. These are chosen based on the limited time scope, which characteristics are believed to be measurable to some degree and are the most relevant for the testbed. Hence, basic rather than extra characteristics are included in the evaluation criteria. Using the terminology of softgoals and hardgoals, the non-functional requirements will be approached as softgoals.

Based on the framework in [20113], the non-functional requirements being used in the evaluation criteria is listed below:

2 Availability: *"degree to which a system, product or component is operational and accessible when required for use."*

3 Functional Completeness: *"degree to which the set of functions covers all the specified tasks and user objectives."*

The nature of the proposed self-healing approaches represents the reasoning for the chosen non-functional requirements. Procedures such as steps 1 and 2 are drastic, therefore we presume that both availability and functional completeness will be affected. Evaluating their performance with regards to these aspects are therefore highly relevant, and are furthermore fairly essential characteristics of operational web servers. Referring to figure 3.1, availability and functional completeness are subcharacteristics of respectively reliability and functional suitability. The remaining six characteristics; performance efficiency, compatibility, usability, security, maintain-ability, and portability, are not prioritized because we believe that these are not as relevant to the task.

The evaluation criteria are summarized below. With the notion of softgoals and hardgoals, the functional requirement is formulated in such a way that the measurement result is binary, whereas the non-functional requirements are formulated such that the measurement results follow a scale.

1 Healing: The self-healing procedure must remove the vulnerability or susceptibility at hand in the testbed, and make it unreachable and/or unexploitable.

2 Availability: To what degree is the web server operational and accessible after being healed compared to before the healing process was initiated?

3 Functional Completeness: To what degree covers the set of functions all the specified tasks and user objectives after being healed compared to before the healing process was initiated?

3.4 Implementation and measurement of self-healing

To get a sense of how well the self-healing techniques perform, they must be measured against the evaluation criteria. This can be done by applying certain types of testing to witness how the changes affected the system. Having three main criteria, these will be addressed individually.

3.4.1 Software testing

Software testing is *"a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended"* [MSB11]. For the purpose of measuring how well each self-healing approach performs, applying such tests gives objective insight to the operation of the web server. There is a variety of tests used in software testing, but the ones with the most relevance for the thesis are further briefly described.

Functional testing

In functional testing, *"tests are constructed from the functional properties of the program that are specified in the program's requirements"* [How80]. The piece of software being tested is treated as a mathematical function, having inputs and corresponding outputs.

Unit testing When testing individual subprograms, subroutines, classes, and procedures in a program, unit testing is performed. Smaller parts of the program is being tested rather than the program as a whole [MSB11].

Integration testing After having each unit or module tested, the parts are combined, and integration testing is applied [LW90]. It focuses on the interaction between the units as a group.

Regression testing If modifications have been made to a program, regression tests ensures that the program performs according to its specification [LW89]. Test cases are used to reveal potential errors that the alterations may have introduced.

Non-functional testing

Non-functional testing tests the behavior of a system or program, rather than specific functional requirements. It assures that the system is in possession of certain non-functional characteristics such as reliability and performance. The results from testing are measurable, meaning that the results are not subjective. Non-functional testing is performed after functional testing.

3.4.2 Healing criteria

The healing evaluation criteria measures whether the self-healing done was successful or not, consequently whether the vulnerability is still present, reachable, and/or exploitable. For this criteria to be measurable, the same attack which caused the vulnerability to be exploited must be relaunched. It is possible to create a unit test, either autonomously or manually, which checks if the same attack causes the same responses from the system. Since each vulnerability and corresponding web page probably has a distinctive behavior, each web page requires individual, specific tests.

3.4.3 Availability criteria

The availability evaluation criteria measures whether the self-healing mechanism has caused downtime on one or more of the web pages. In order to check this, it is possible to run a script that checks the availability of each vulnerable page. If the page is available, the HTTP response code[15] is `OK 200`, whereas it would be `40X` if it is unavailable.

3.4.4 Functional Completeness criteria

The functional completeness criteria measures to what degree the essential functionality of the web page is preserved. To enable this, a regression test must be made in

[15]HTTP response status codes indicate whether a HTTP request was successful.

advance of the healing process, checking if the functionality is in accordance with the specification. Hence, each path on the web server to be tested must have their own, individual test, ensuring correct behavior.

Chapter 4 Experimental Setup

By the term *"experimental setup"* we understand the tools, platforms and settings used to perform the practical experiments of the project. In pursuance of the most appropriate setup, several appliances should be considered and evaluated according to set criteria. In this chapter, the process of deriving the experimental setup, including platforms, tools, and settings used in the experiments as well as details of the setup used, will be presented. The specifications of the machines used throughout the project are listed in table 4.1 and 4.2.

OS	Ubuntu 18.04.4 LTS
Memory	31,3 GiB
Processor	Intel®Core™i7-4790 CPU @ 3.60GHz x 8
Graphics	Intel®Haswell Desktop
GNOME	3.28.2
OS type	64-bit
Disk	503,0 GB

Table 4.1: Machine specifications, Ubuntu.

OS	macOS Catalina 10.15.3
Memory	8 GB 1867 MHz DDR3
Processor	2,7 GHz Dual-Core Intel Core i5
Graphics	Intel Iris Graphics 6100 1536 MB
OS type	64-bit
Disk	250,8 GB

Table 4.2: Machine specifications, macOS.

4.1 Selection of tools

Selection of tools includes the process of selecting the most optimal appliances for a project based on the aims of the project. For this thesis, investigations of self-healing web servers written in PHP are conducted, and it is therefore required both a test server in which to heal and automated tools to find vulnerabilities to patch. Evaluation criteria and methodology for validating testbeds and scanners are further described, respectively, in sections 3.2.1 and 3.2.2.

4.1.1 Testbed

Two alternatives have been evaluated for selecting the testbed; developing the testbed by oneself or implementing an open-source web server repository.

Developing the testbed

Having the testbed custom made for the project ensures flexibility towards modifications and management. It allows for implementations of numerous web vulnerabilities with a diversity of flexibility levels. The platform can be written in any programming language, hence PHP. However, implementing this alternative is expected to be more time demanding than the option of using pre-made solutions.

Implementing existing solutions

Using pre-made solutions might provide less diversity in present vulnerabilities, the programming language could differ from PHP, and the integration could be more troublesome. Despite these drawbacks, the option is significantly better concerning the time limitations.

The *Damn Vulnerable Web Application (DVWA)*[1] is a PHP/MySQL based web application developed with intentional web vulnerabilities. It is created for educational purposes, where the user can both learn about and teach common web vulnerabilities. The platform is available in Docker, has a variety of vulnerabilities, it is straightforward to implement and is very well documented. The user may choose between four difficulty levels; low, medium, high, and impossible, whereas in this thesis, low is the most appropriate level. The solution comes with an IDS (PHPIDS[2]), and also has the advantage of being time-efficient since it is a complete product.

As documented in the pre-project (section 1.2) of the thesis, it was early in the process suggested that the DVWA could be an appropriate choice for a testbed.

[1]http://www.dvwa.co.uk
[2]https://github.com/PHPIDS/PHPIDS

4.1.2 Scanners

In the following subsection, results from testing potential vulnerability scanners for the thesis are presented. The tools are phpcs-security-audit, RIPS and SonarQube with SonarScanner.

phpcs-security-audit

The tool is a set of PHP_CodeSniffer[3] rules that finds security weaknesses in PHP source code. The tool appears to be regularly maintained and is tolerably documented. It is only command line accessible and is fairly easy to set up and implement. With regards to cost and availability, the tool is open-source and therefore free to use for an unlimited amount of scans. Its coverage is decent, being able to indicate and detect vulnerabilities such as insecure SQL queries, use of file upload functions with direct user input, and possible XSSs. The format of created vulnerability logs is fairly documented, including which file raised the error or warning, which line number in the code contains weaknesses and what kind of vulnerability is indicated. Listing 4.1 displays an example output from the scanner.

```
FILE: /var/www/html/DVWA/vulnerabilities/csrf/source/low.php
----------------------------------------------------------------
FOUND 0 ERRORS AND 8 WARNINGS AFFECTING 5 LINES
----------------------------------------------------------------
  3 | WARNING | User input detetected with $_GET.
  5 | WARNING | User input detetected with $_GET.
  6 | WARNING | User input detetected with $_GET.
 12 | WARNING | Crypto function md5 used.
 16 | WARNING | MYSQLi function mysqli_query() detected with dynamic
    parameter
 16 | WARNING | HTML construction with ( detected.
 16 | WARNING | Possible XSS detected with . on die
 16 | WARNING | HTML construction with mysqli_connect_error() detected.
----------------------------------------------------------------


FILE: /var/www/html/DVWA/vulnerabilities/csrf/source/high.php
----------------------------------------------------------------
FOUND 0 ERRORS AND 9 WARNINGS AFFECTING 6 LINES
----------------------------------------------------------------
  3 | WARNING | User input detetected with $_GET.
  5 | WARNING | User input detetected with $_REQUEST.
  8 | WARNING | User input detetected with $_GET.
  9 | WARNING | User input detetected with $_GET.
 15 | WARNING | Crypto function md5 used.
 19 | WARNING | MYSQLi function mysqli_query() detected with dynamic
    parameter
```

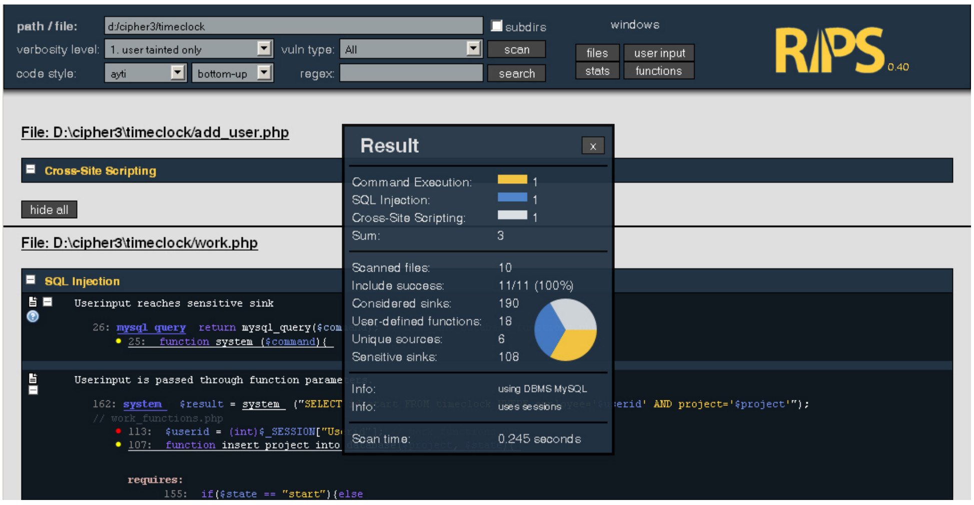
[3]https://github.com/squizlabs/PHP_CodeSniffer

```
19 | WARNING | HTML construction with ( detected.
19 | WARNING | Possible XSS detected with . on die
19 | WARNING | HTML construction with mysqli_connect_error() detected.
---------------------------------------------------------------
```

Listing 4.1: Example output from phpcs-security-audit.

RIPS

The RIPS static source code analyzer offers two generations of scanners. The first generation (RIPS 0.5), which has been tested in this project, is an open-source and free version, whereas the new generation is directed towards commercial use with greater functionality. The development of RIPS 0.5 has been abandoned since 2013, and this generation does not support scans using the command line. Since it is not maintained, the implementation was troublesome, and documentation as of today appears outdated. Figure 4.1 illustrates the interface of the RIPS scanner.



path / file: d:/cipher3/timeclock
verbosity level: 1. user tainted only
vuln type: All
code style: ayti
bottom-up

File: D:\cipher3\timeclock/add_user.php

Cross-Site Scripting

File: D:\cipher3\timeclock/work.php

SQL Injection

```
Userinput reaches sensitive sink

    26: mysql query  return mysql_query($com
      • 25:  function system ($command){

Userinput is passed through function parame

    162: system  $result = system ("SELECT
    // work_functions.php
      • 113:  $userid = (int)$_SESSION["Us
      • 107:  function insert project into

      requires:
          155:  if($state == "start"){else
```

Result

Command Execution:	1
SQL Injection:	1
Cross-Site Scripting:	1
Sum:	3
Scanned files:	10
Include success:	11/11 (100%)
Considered sinks:	190
User-defined functions:	18
Unique sources:	6
Sensitive sinks:	108
Info:	using DBMS MySQL
Info:	uses sessions
Scan time:	0.245 seconds

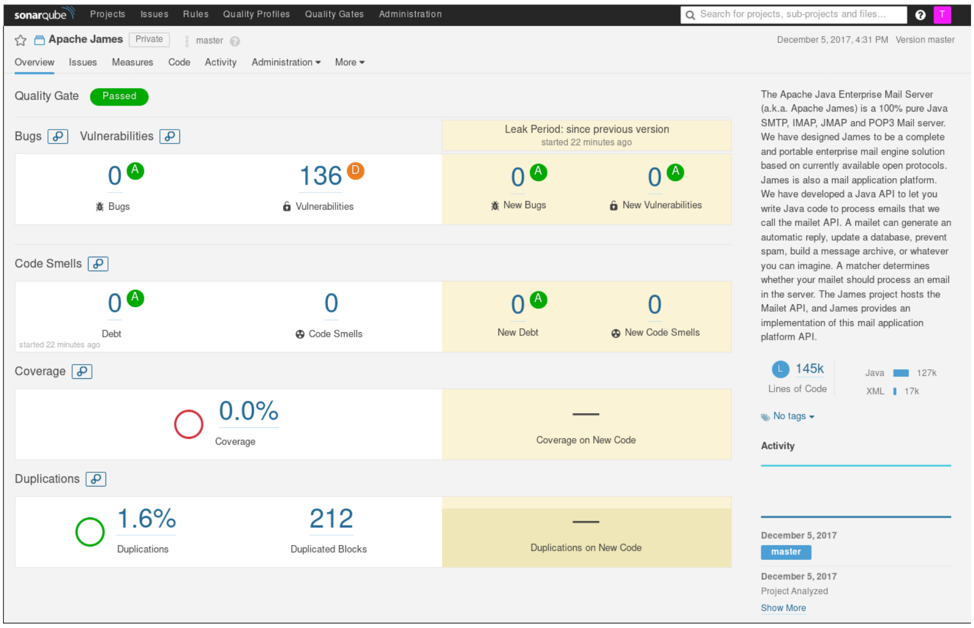
Figure 4.1: Example of RIPS interface[4].

SonarQube with SonarScanner

SonarQube displays the results from the scan in a systematic way, separating between bugs, vulnerabilities and code smells, among other observations. There are four plans available, whereas the free and open-source community edition is used during the selection of tools. Several features are not available in this plan, including command

[4]http://rips-scanner.sourceforge.net/stats.jpg

line support and being able to extract security logs into files. These drawbacks have the potential of making automation, integration, and management significantly more challenging. Figure 4.2 displays the user interface of the SonarScanner.



December 5, 2017, 4:31 PM Version master

Quality Gate Passed

Bugs Vulnerabilities

Leak Period: since previous version
started 22 minutes ago

0 A Bugs	136 D Vulnerabilities	0 A New Bugs	0 A New Vulnerabilities

Code Smells

0 A Debt	0 Code Smells	0 A New Debt	0 New Code Smells

started 22 minutes ago

Coverage

0.0% Coverage	— Coverage on New Code

Duplications

1.6% Duplications	212 Duplicated Blocks	— Duplications on New Code

The Apache Java Enterprise Mail Server (a.k.a. Apache James) is a 100% pure Java SMTP, IMAP, JMAP and POP3 Mail server. We have designed James to be a complete and portable enterprise mail engine solution based on currently available open protocols. James is also a mail application platform. We have developed a Java API to let you write Java code to process emails that we call the mailet API. A mailet can generate an automatic reply, update a database, prevent spam, build a message archive, or whatever you can imagine. A matcher determines whether your mailet should process an email in the server. The James project hosts the Mailet API, and James provides an implementation of this mail application platform API.

L 145k Lines of Code
Java 127k
XML 17k

No tags

Activity

December 5, 2017
master

December 5, 2017
Project Analyzed

Show More

Figure 4.2: Example of SonarQube with SonarScanner interface[5].

4.1.3 A summary for used tools

This subsection summarizes the use of testbeds and scanners as tools for this master thesis in two tables: table 4.3 and 4.4. Even though neither candidates appear optimal, some are more suitable than others. For the evaluation of testbeds, choosing an already existing solution becomes the prevalent choice, mostly because of the limited time frame. With regard to the scanners, there are several options with multiple drawbacks and advantages. The options which have an additional premium, enterprise, or next-generation version, often score worse than those who do not. Based on the evaluation criteria, phpcs-security-audit is the best-fitted candidate out of the scanners. However, even the scanners which perform the best with regards to coverage are unable to detect slightly more complex vulnerabilities such as CSRF and brute force prone forms.

[5]https://miro.medium.com/max/1400/1*ZrnUXbYQk0881qcLuNYC4g.png

Other potential scanners, not tested in this thesis, are progpilot[6], graudit[7], and VisualCodeGrepper[8]. These were not tested due to time constraints and unforeseen changes in accessible machine equipment.

Platform	**Integratable**	**Variety of vulnerabilities**	**PHP based**	**Time efficient**	**Limitations**
Developing the testbed	Yes	Yes	Yes	No	Might require a large amount of time to develop and implement
Implementing existing solutions; DVWA	Yes	Yes	Yes	Yes	Less flexible than a self-developed solution

Table 4.3: Evaluation matrix of testbeds.

Tool	**Coverage**	**Integration and management**	**Cost**	**PHP support**	**Comments**
phpcs-security-audit	Satisfying	Satisfying	Open-source	Yes	- False positives occur
RIPS	Satisfying	Not satisfying	Open-source	Yes	- Only GUI - Not maintained
SonarQube with sonar-scanner	Satisfying	Not satisfying	Open-source	Yes	- Only GUI - Results from scans not possible to extract to file for further processing

Table 4.4: Evaluation matrix of scanners.

4.2 Set up of tools during testing

In this section, an overview of the technical set up is given including the tools we have selected. Briefly, DVWA is run in a Docker container. The PHPIDS is constantly enabled, and is the only source of breaches used in the self-healing script. The vulnerability scanner, phpcs-security-audit, is launched for certain self-healing procedures.

DVWA with PHPIDS The test server is always set to difficulty level `"low"`. The IDS is always enabled, except for when we do not want the IDS to block the incoming attack. The DVWA runs in a container, and is launched with the following command: `docker run -rm -it -p 80:80 -name <name> dvwa:<latest tag>`.

[6]https://github.com/designsecurity/progpilot
[7]https://github.com/wireghoul/graudit/
[8]https://github.com/nccgroup/VCG

phpcs-security-audit The vulnerability scanner is installed in the Docker container and is launched using the command: `php PHP_CodeSniffer/bin/phpcs -extensions=php,inc,lib,module,info -standard=/root/path/to/ruleset \ /phpcs-security-audit/example_drupal7_ruleset.xml /root/path/to/files`. There are two pre-existing rulesets, and for the thesis, ruleset `example_drupal7_ruleset.xml` is used.

Even though the tools allows for modifications to the rulesets, changes have not been made to either.

Container specifications The DVWA is installed as a Docker container, following the instructions from the developers[9]. In addition to the pre-existing packages on the container, the following list contains additional packages installed, excluding Python libraries:

- Python 2
- pip
- PHP_CodeSniffer
- phpcs-security-audit
- nano
- curl

4.3 Overview of code

In this section, we present the script used (appendix A.1) to conduct self-healing and explore self-healing capabilities. A brief explanation of the different components comprising the script is included. This excludes the functionality of the five self-healing procedures, which will be further delved into in section 5.1, 5.2, 5.3, 5.4 and 5.5. Figure 4.3 illustrates how the components interact with each other.

4.3.1 Detecting and responding to breaches

The self-healing procedure is initiated when the IDS detects an attack by logging it in the log file, illustrated in figure 4.4. This is realized by monitoring `phpids_log.txt` of PHPIDS for changes, hence if a security incident has taken place. PHPIDS is implemented in DVWA, and is therefore used as the IDS in this project.

[9]https://github.com/ethicalhack3r/DVWA

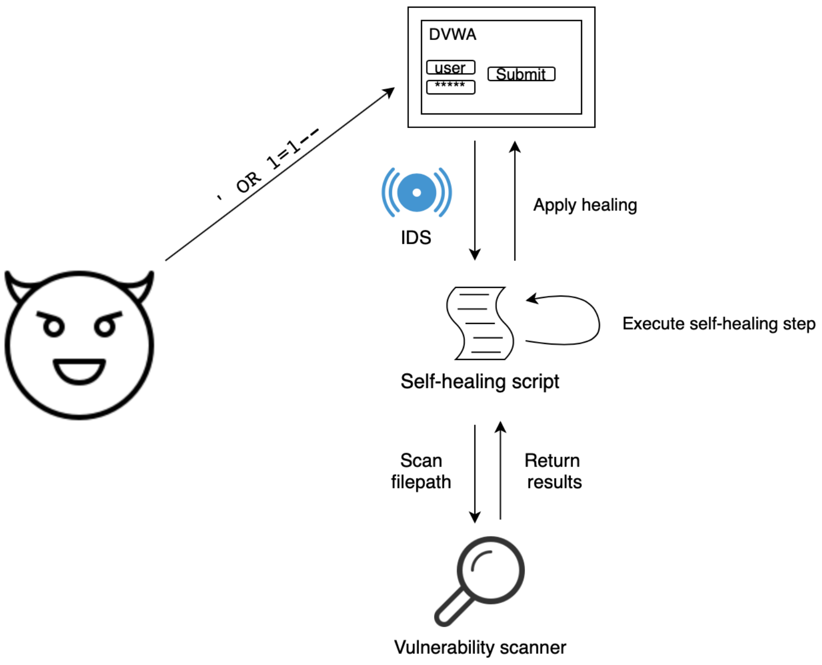
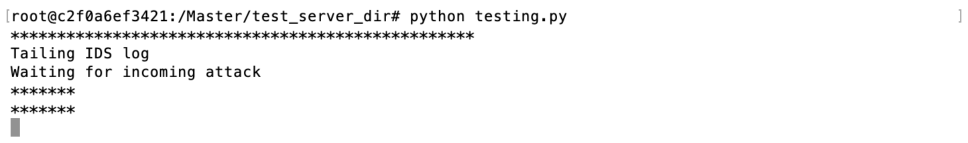


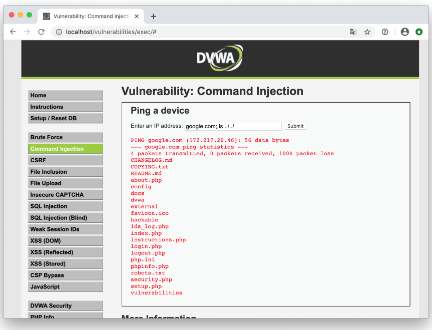
Figure 4.3: Figure of self-healing script and its components. The IDS monitors the server, and whenever events happen that trigger the IDS, the script is notified. One of the self-healing procedures is chosen, and the vulnerability scanner is potentially invoked. The result from the healing technique is applied to the server.



```
root@c2f0a6ef3421:/Master/test_server_dir# python testing.py
***************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
```

Figure 4.4: Monitoring the IDS log for new attacks.

If the IDS is disabled, attacks and exploits will not get caught, since the IDS is the only source of breaches in the system. Figure 4.5 illustrates a command injection exploit with its corresponding response when the IDS is disabled.



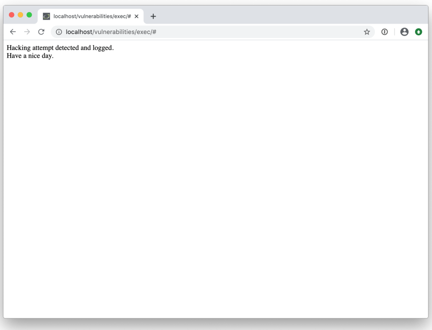
Vulnerability: Command Injection

Ping a device

Enter an IP address: google.com; ls ../../

```
PING google.com (172.217.20.46): 56 data bytes
--- google.com ping statistics ---
4 packets transmitted, 0 packets received, 100% packet loss
CHANGELOG.md
COPYING.txt
README.md
about.php
config
docs
dvwa
external
favicon.ico
hackable
ids_log.php
index.php
instructions.php
login.php
logout.php
php.ini
phpinfo.php
robots.txt
security.php
setup.php
vulnerabilities
```

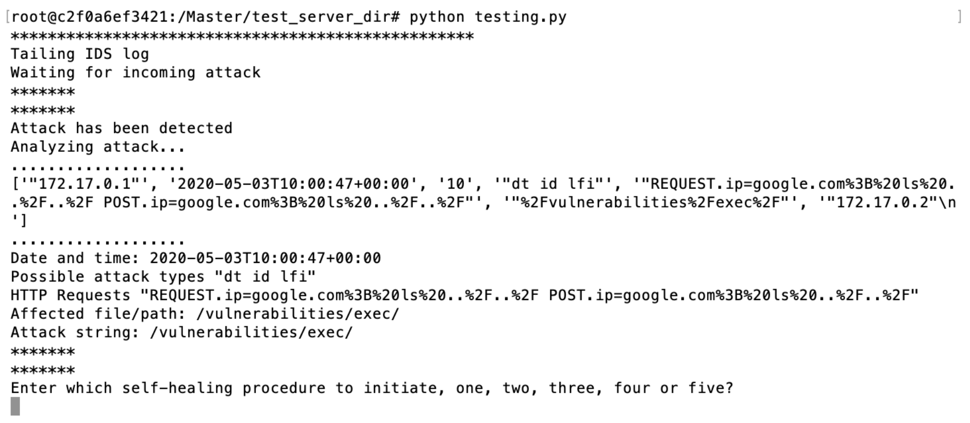
Figure 4.5: Command injection vulnerability example in DVWA.



Hacking attempt detected and logged.
Have a nice day.

Figure 4.6: Message displayed when an attack attempt has been detected.

However, if the IDS is enabled, the browser will demonstrate that the attack has been detected, and the self-healing process will start. Using the same attack launched in figure 4.5, the response from the browser and the script are illustrated in figure 4.6 and figure 4.7. At this point, the script expects the administrator to choose which of the five self-healing techniques (3.3.1) to handle the incident.



```
root@c2f0a6ef3421:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-03T10:00:47+00:00', '10', '"dt id lfi"', '"REQUEST.ip=google.com%3B%20ls%20.
.%2F..%2F POST.ip=google.com%3B%20ls%20..%2F..%2F"', '"%2Fvulnerabilities%2Fexec%2F"', '"172.17.0.2"\n
']
...................
Date and time: 2020-05-03T10:00:47+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.ip=google.com%3B%20ls%20..%2F..%2F POST.ip=google.com%3B%20ls%20..%2F..%2F"
Affected file/path: /vulnerabilities/exec/
Attack string: /vulnerabilities/exec/
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
```

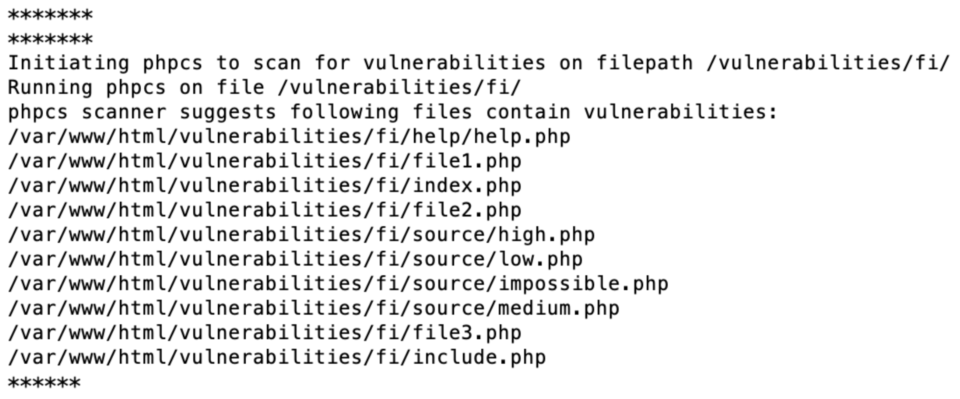
Figure 4.7: Response to hacking attempt in the self-healing script.

4.3.2 Using PHPIDS and phpcs-security-audit scanner

The PHPIDS is not only used to detect the breach, but assists in determining the cause of the breach. phpcs-security-audit uses the result from the PHPIDS in order to scan for vulnerabilities. The scanner is only invoked for steps 2, 3, 4, and 5. An example of a log entry in PHPIDS, phpcs-security-audit output in the self-healing script and parts of the corresponding phpcs-security-audit result are given in respectfully listing 4.2, figure 4.8 and listing 4.3.

```
"172.17.0.1",2020-05-05T17:08:35+00:00,20,"dt id lfi","REQUEST.page
    =..%2F..%2F..%2F..%2Fetc%2Fpassw GET.page=..%2F..%2F..%2F..%2Fetc%2
    Fpassw","%2Fvulnerabilities%2Ffi%2F%3Fpage%3D..%2F..%2F..%2F..%2
    Fetc%2Fpassw","172.17.0.2"
```

Listing 4.2: Example entry in PHPIDS log from a directory traversal attempt.



```
*******
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/fi/
Running phpcs on file /vulnerabilities/fi/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/fi/help/help.php
/var/www/html/vulnerabilities/fi/file1.php
/var/www/html/vulnerabilities/fi/index.php
/var/www/html/vulnerabilities/fi/file2.php
/var/www/html/vulnerabilities/fi/source/high.php
/var/www/html/vulnerabilities/fi/source/low.php
/var/www/html/vulnerabilities/fi/source/impossible.php
/var/www/html/vulnerabilities/fi/source/medium.php
/var/www/html/vulnerabilities/fi/file3.php
/var/www/html/vulnerabilities/fi/include.php
******
```

Figure 4.8: Example output form phpcs-security-scanner in the self-healing script.

```
FILE: /var/www/html/vulnerabilities/fi/source/high.php
--------------------------------------------------------------------------------
FOUND 0 ERRORS AND 2 WARNINGS AFFECTING 2 LINES
--------------------------------------------------------------------------------
 4 | WARNING | User input detetected with $_GET.
 7 | WARNING | Filesystem function fnmatch() detected with dynamic
    parameter
--------------------------------------------------------------------------------
```

Listing 4.3: Parts of phpcs-security-audit scanning results.

When using phpcs-security-audit, the output is saved to a .txt-file, and reshaped to a format which can be used for later. The code for calling phpcs-security-audit and handling its output from the self-healing script are presented in listing 4.4.

```
def phpcs(filepath,date):
  print "*******"
  print "*******"
  print("Initiating phpcs to scan for vulnerabilities on filepath "+
    filepath)

  command = "php PHP_CodeSniffer/bin/phpcs --extensions=php,inc,lib,
    module,info --standard=/Master/test_server_dir/phpcs-security-audit
    /example_drupal7_ruleset.xml /var/www/html"+filepath
  command_list =  command.split(" ")
  filepath_output = "/Master/test_server_dir/phpcs_outputs/"
  filename = filepath[len("vulnerabilities")+1:]+"_"+date[:19]+".txt"
  filename = filename.replace("/", "")
  f = open(filepath_output+filename,"w")

  subprocess.call(command_list, stdout=f)

  return filepath_output+filename

def phpcs_file_loc(phpcs_results_txt):
    fp = open(phpcs_results_txt, 'r')
  files_loc=[]
  while True:
    line = fp.readline()
                locs_found=[]
                full_error_message = []

    if line[0:4] == "FILE":
      vuln_file = line[6:].rstrip("\n")
      while True:
        locs=fp.readline()
        if len(locs) > 1:
          if locs[1].isdigit() == True or locs[2].isdigit() == True or
    locs[3].isdigit() == True:
            if int(filter(str.isdigit, locs[0:4])) not in locs_found:
              locs_found.append(int(filter(str.isdigit, locs[0:4])))
                                        full_error_message.append(locs)
              continue
        elif len(locs) == 1:
          files_loc.append([vuln_file,locs_found,full_error_message])
          vuln_file =""
          locs_found=[]
          full_error_message=[]
          break
    if line[0:5] == "Time:":
      break
  fp.close()

```

```
    return files_loc
```

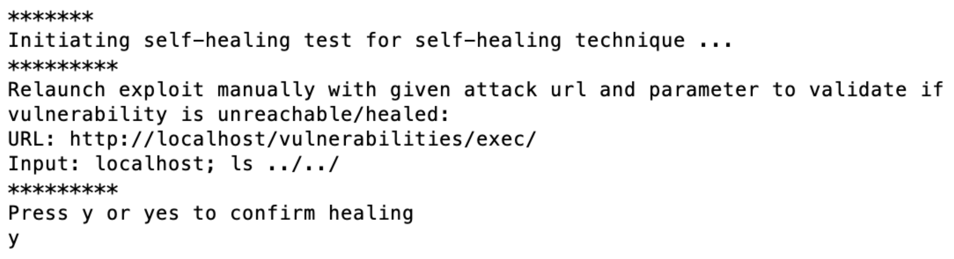
Listing 4.4: Code functionality for executing and handling vulnerability scanner phpcs-security-audit.

4.3.3 Evaluating the self-healing techniques

The self-healing techniques' performance will be evaluated based on three criteria: self-healing, availability, and functionality, which is explained in depth in subsection 3.4. The practical solution to their implementation is further elaborated.

Self-healing evaluation

For the self-healing evaluation, measurements of whether the vulnerability in question is still reachable or present are conducted. This is currently done manually, having the user relaunch the attack which triggered the IDS initially. Based on the results from the exploit, the user inputs to the script whether the attack is still possible to launch successfully. The visual display, when validating the self-healing criteria, is shown in figure 4.9.



```
*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameter to validate if
vulnerability is unreachable/healed:
URL: http://localhost/vulnerabilities/exec/
Input: localhost; ls ../../
*********
Press y or yes to confirm healing
y
```

Figure 4.9: Self-healing test in self-healing script.

Availability evaluation

In order to discover the status of the web pages, that is if they are "up" or "down", scripts in listing 4.5 and 4.6 are run before and after the self-healing process takes place. The bash scripts use `curl` to query the web server using URLs from `path_to_files.txt`, and write the results to respectfully `http_resp_before_heal.txt` and `output_http_resp.txt`.

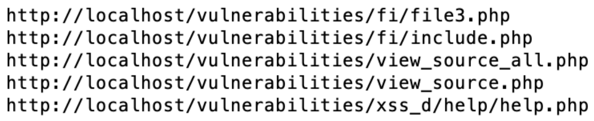
```
for URL in `cat avail_crit/path_to_files.txt`; do echo $URL; curl -m 10
  -LIs $1 "$URL" | grep HTTP/1. |  awk {'print $2'}; done | tee
 avail_crit/http_resp_before_heal.txt
```

Listing 4.5: `get_HTTP_resp_before.sh`

```
for URL in `cat avail_crit/path_to_files.txt`; do echo $URL; curl -m 10
  -LIs $1 "$URL" | grep HTTP/1. |  awk {'print $2'}; done | tee
 avail_crit/output_http_resp.txt
```

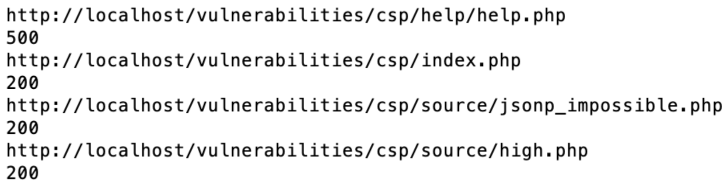
Listing 4.6: `get_HTTP_resp.sh`

Content of `path_to_files.txt` and `http_resp_before_heal.txt`/`output_http_resp.txt` are exemplified in snippets in respectfully figure 4.10 and 4.11.



```
http://localhost/vulnerabilities/fi/file3.php
http://localhost/vulnerabilities/fi/include.php
http://localhost/vulnerabilities/view_source_all.php
http://localhost/vulnerabilities/view_source.php
http://localhost/vulnerabilities/xss_d/help/help.php
```

Figure 4.10: Example of content in `path_to_files.txt`



```
http://localhost/vulnerabilities/csp/help/help.php
500
http://localhost/vulnerabilities/csp/index.php
200
http://localhost/vulnerabilities/csp/source/jsonp_impossible.php
200
http://localhost/vulnerabilities/csp/source/high.php
200
```

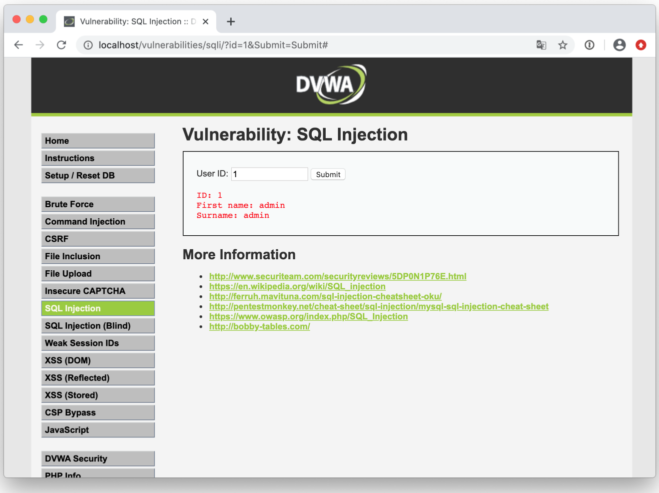
Figure 4.11: Example of content in files containing HTTP response codes.

Using the HTTP response code, the availability test is able to determine if the site is up or down. A site is considered down if the response is "404 Not found". Due to redirection, the HTTP response code might include "302 Found" in addition to "200 OK". This is removed from the text files since they are irrelevant to the task. The availability test returns the number of pages in total, number of pages down before healing, and number of pages down after healing. The complete code of the availability test is available in appendix A.1.

Functionality evaluation

The third and final evaluation criteria measures to what extent the functionality of the healed page is kept. Unit tests for the relevant scenarios must be created in advance, and for this project, tests are created for `/vulnerabilities/sqli/`, `/vulnerabilities/fi/` and `/vulnerabilities/exec/`. These are run autonomously after healing, and the results from testing are presented. If a file is given, which is neither of the three, the test reports "**None**" as the result. Following, the functionality tests of the three files are presented.

∗`/vulnerabilities/sqli/` The page has the functionality of displaying in the browser the given user ID. The test therefore checks whether the output is as given in figure 4.15 when using the input **`"1"`**.



Vulnerability: SQL Injection

User ID: 1

```
ID: 1
First name: admin
Surname: admin
```

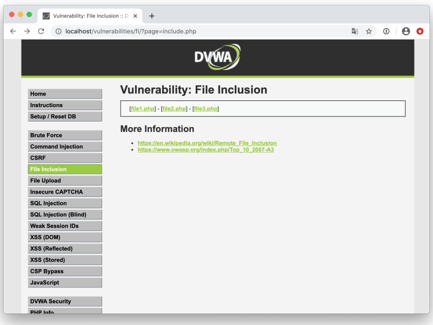
More Information

- http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- https://en.wikipedia.org/wiki/SQL_injection
- http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
- https://www.owasp.org/index.php/SQL_Injection
- http://bobby-tables.com/

Figure 4.12: Expected output in functionality test of `/vulnerabilities/sqli/`.

The evaluation score is **1** if it is able to display the given user ID after healing.

∗`/vulnerabilities/fi/` The page shall include either one of three pages which the user chooses. The test therefore tests if the given page is included and if the page gives the correct output. The browser display is illustrated in figure 4.13 and 4.14.



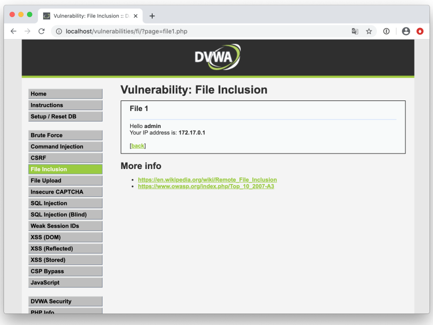
Vulnerability: File Inclusion

[file1.php] - [file2.php] - [file3.php]

More Information

- https://en.wikipedia.org/wiki/Remote_File_Inclusion
- https://www.owasp.org/index.php/Top_10_2007-A3

Figure 4.13: Expected output in functionality test of `/vulnerabilities/fi/`.



Vulnerability: File Inclusion

File 1

Hello admin
Your IP address is: 172.17.0.1

[back]

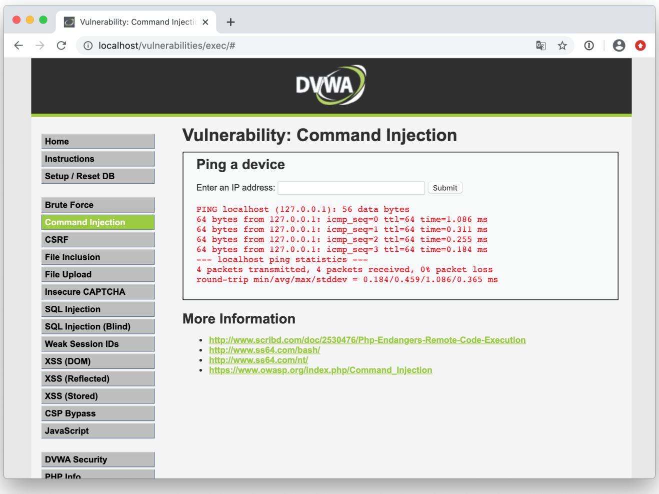
More info

- https://en.wikipedia.org/wiki/Remote_File_Inclusion
- https://www.owasp.org/index.php/Top_10_2007-A3

Figure 4.14: Expected output in functionality test of `/vulnerabilities/fi/` when choosing `file1.php`.

The evaluation score is **1** if it is able to display all three files with expected output after healing.

∗`/vulnerabilities/exec/` The page is expected to `ping` a given URL. Hence, the functionality test checks if the given URL is queried. An example output from the page with expected behaviour is given in figure 4.15.



Vulnerability: Command Injection

Ping a device

Enter an IP address:

```
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=1.086 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.311 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.255 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.184 ms
--- localhost ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.184/0.459/1.086/0.365 ms
```

More Information

- http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
- http://www.ss64.com/bash/
- http://www.ss64.com/nt/
- https://www.owasp.org/index.php/Command_Injection

Figure 4.15: Expected output in functionality test of `/vulnerabilities/exec/`.

The evaluation score is **`1`** if it is able `ping` the given URL after healing. The full code is available in appendix A.1.

Chapter 5

Experiments and Results

In this chapter, the results from developing a self-healing script and using it to autonomously remove vulnerabilities are presented. The system uses an IDS to detect intrusions and a vulnerability scanner to find susceptibilities in the code. Five self-healing "steps" have been developed, whereas the first is the most radical approach and the last technique is more nuanced and precise towards particular vulnerabilities. In order to measure the efficiency of each self-healing procedure, three evaluation criteria is created.

For each step, the results are summarized in a table. Each experiment is given a score for how well they satisfy the three evaluation criteria, and the average of all the experiments of the current step are presented in a row named **"Average"**. However, vulnerabilities in experiments not covered in certain tests, such as the functionality test, are given the score **"None"**. When producing the average of all experiments in such a scenario, the functionality will be documented manually and be included in the calculation based on these manual findings. A corresponding table is given at the end of the chapter as well, though it summarizes the performance of each healing technique.

5.1 Step 1 - Power off web server

The self-healing approach tries to remove or make the vulnerability unreachable by taking down the web server when an attack has been detected. The code snippet for the procedure is included in listing 5.1.

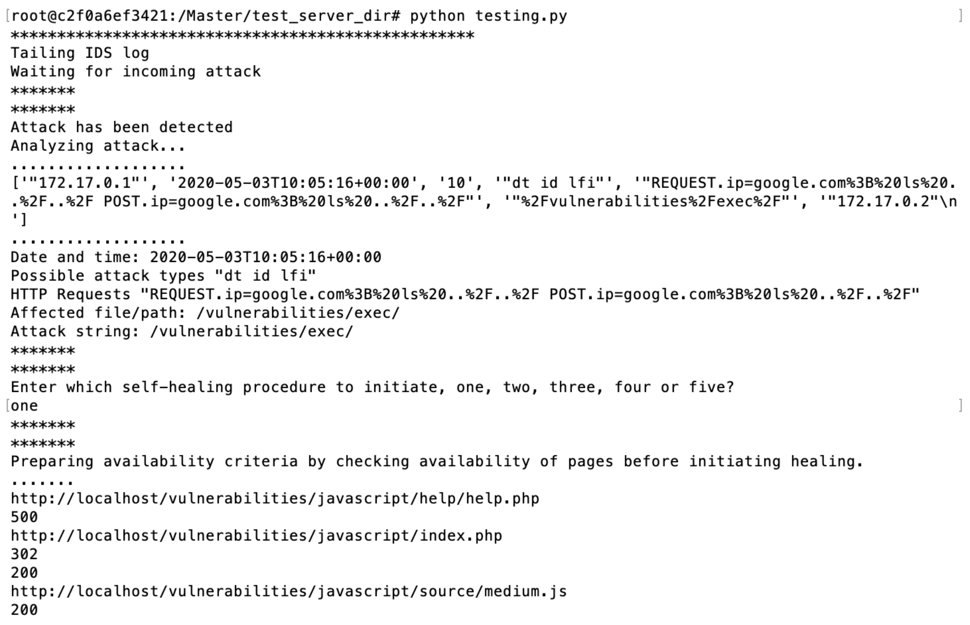
```python
def step1():
        print("*******")
        print("Self-healing mechanism: turn of server")
        print("Server will shutdown now")
        #launch stop server script
        os.system("./stop_server.sh")
        print("*******")
```

Listing 5.1: Code snippet of self-healing technique step 1.

A bash script is used in the code of step 1 to turn off the server, presented in listing 5.2.

```bash
#!/bin/bash
echo "stop server"
stopcommand="service apache2 stop"
eval $stopcommand
```

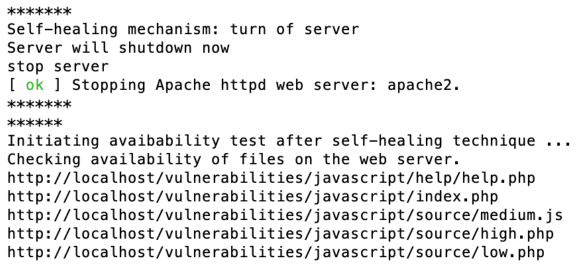
Listing 5.2: `stop_server.sh` script used in step 1.



```
root@c2f0a6ef3421:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-03T10:05:16+00:00', '10', '"dt id lfi"', '"REQUEST.ip=google.com%3B%20ls%20.
.%2F..%2F POST.ip=google.com%3B%20ls%20..%2F..%2F"', '"%2Fvulnerabilities%2Fexec%2F"', '"172.17.0.2"\n
']
...................
Date and time: 2020-05-03T10:05:16+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.ip=google.com%3B%20ls%20..%2F..%2F POST.ip=google.com%3B%20ls%20..%2F..%2F"
Affected file/path: /vulnerabilities/exec/
Attack string: /vulnerabilities/exec/
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
one
*******
*******
Preparing availability criteria by checking availability of pages before initiating healing.
.......
http://localhost/vulnerabilities/javascript/help/help.php
500
http://localhost/vulnerabilities/javascript/index.php
302
200
http://localhost/vulnerabilities/javascript/source/medium.js
200
```

Figure 5.1: Command injection detected and step 1 is chosen as the self-healing technique.

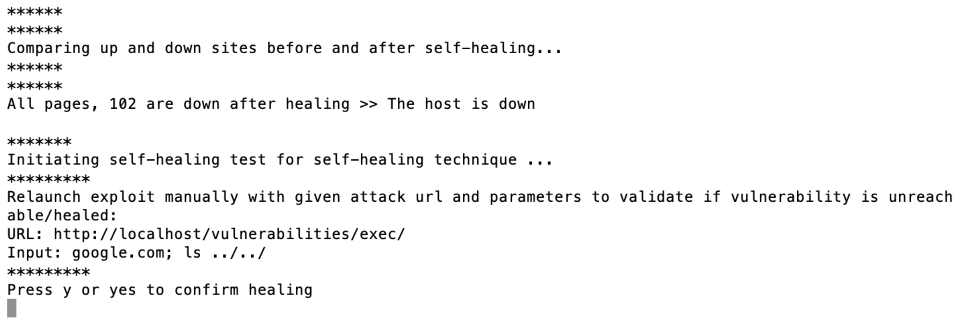
When testing the self-healing abilities of step 1, an attack is launched towards the server in which the IDS must detect. In the following example, a command injection is performed using input **"google.com; ls ../../"**, equivalent to that presented in figure 4.5 and figure 4.7. After having executed the attack, the response from the self-healing script is showed in figure 5.1 when choosing step "one" as the self-healing procedure. As described in subsection 4.3.3, the availability measurement process is initiated at once. After the necessary pretests, the self-healing is initiated, shown in figure 5.2, where the server is shut down.



```
*******
Self-healing mechanism: turn of server
Server will shutdown now
stop server
[ ok ] Stopping Apache httpd web server: apache2.
*******
******
Initiating avaibability test after self-healing technique ...
Checking availability of files on the web server.
http://localhost/vulnerabilities/javascript/help/help.php
http://localhost/vulnerabilities/javascript/index.php
http://localhost/vulnerabilities/javascript/source/medium.js
http://localhost/vulnerabilities/javascript/source/high.php
http://localhost/vulnerabilities/javascript/source/low.php
```

Figure 5.2: Self-healing initiated and availability test launched in step 1.

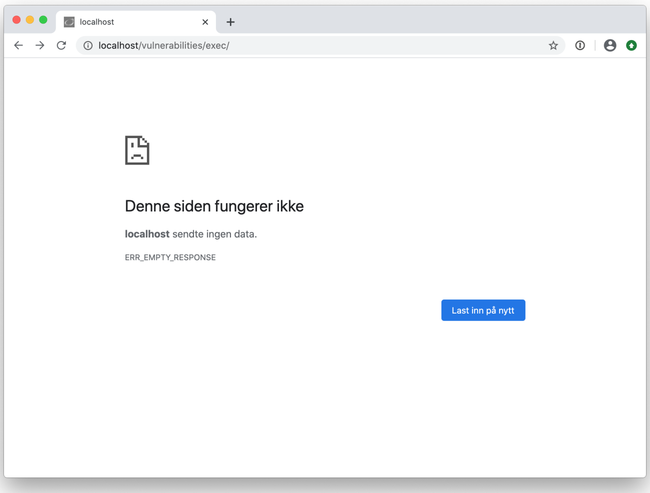
When the self-healing has taken place, the three tests are launched. Both the availability and the self-healing criteria test is launched, demonstrated in figure 5.3. In the latter, the administrator checks manually if the vulnerability is still present, hence in this situation, if the output is equivalent in the browser as that presented in figure 4.5.



```
******
******
Comparing up and down sites before and after self-healing...
******
******
All pages, 102 are down after healing >> The host is down

*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameters to validate if vulnerability is unreach
able/healed:
URL: http://localhost/vulnerabilities/exec/
Input: google.com; ls ../../
*********
Press y or yes to confirm healing
```

Figure 5.3: Availability test and self-healing test in step 1.

As expected, when trying to confirm whether the vulnerability is removed by using the output from the self-healing test, the server is down. The scenario is displayed in figure 5.5.



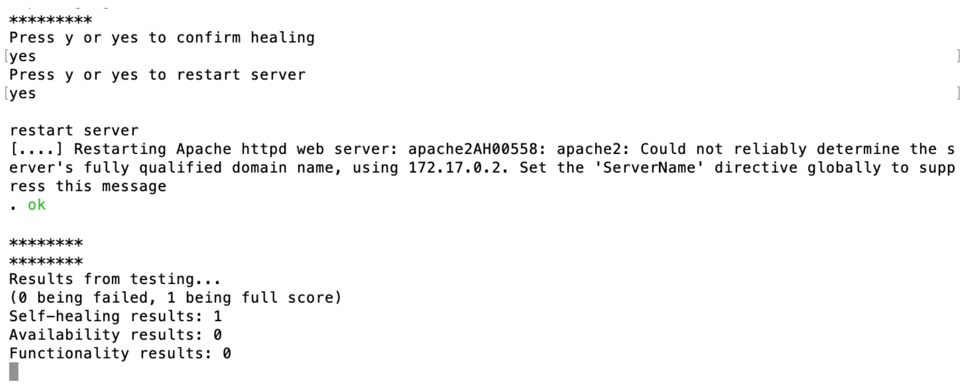
Denne siden fungerer ikke

localhost sendte ingen data.

ERR_EMPTY_RESPONSE

Figure 5.4: Result from self-healing test in step 1.

Note, the functionality test is not run in this scenario, since the server is down. After having performed self-healing and run the tests, the results are presented, as illustrated in figure 5.5.



```
*********
Press y or yes to confirm healing
yes
Press y or yes to restart server
yes

restart server
[....] Restarting Apache httpd web server: apache2AH00558: apache2: Could not reliably determine the s
erver's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to supp
ress this message
. ok

********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 0
Functionality results: 0
```

Figure 5.5: Results from running and testing self-healing technique step 1.

Summary of results

The technique is able to manage the vulnerability by making it unreachable, and hence passes the self-healing test. Since the server as a consequence is taken down, both availability and functionality tests perform poorly since the web server is inaccessible. It was not necessary to run multiple exploits for this self-healing process, since it conducts healing uniformly - the server is shut down whenever a breach is detected regardless of what kind of attack it is. There is no variety to the technique.

	Self-healing test	Availability test	Functionality test	Average
Step 1	1	0	0	0.333

Table 5.1: Summary of results for step 1. The self-healing test achieved full score, whereas the availability and functionality test performed poorly.

5.2 Step 2 - Remove susceptible PHP file

In this self-healing procedure, the PHP file(s) suspected of being vulnerable is removed from the server. There were conducted three distinct experiments, trying to heal a SQL injection, file inclusion, and command injection vulnerability.

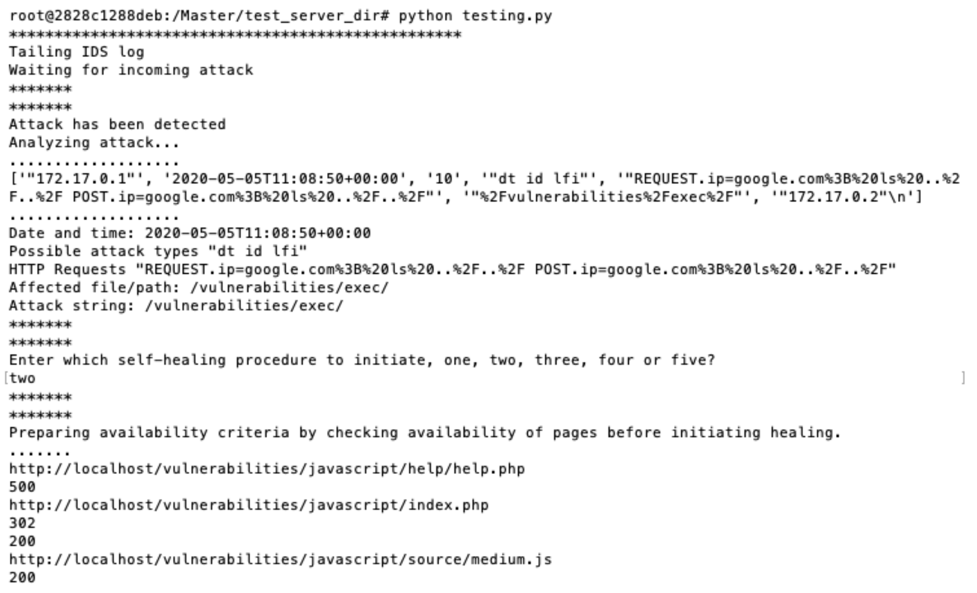
```
def step2(filepath,vuln_files):
        #filepath: path from PHPIDS
        #vulnfiles: full filepath of files which the vulnerability scanner indicated to be vulnerable
        print("*******")
        print("Self-healing mechanism: remove vulnerable PHP file")
        print("*******")
        print("When scanning "+filepath+", phpcs suggests following files are vulnerable: "+ ', '.join(vuln_files))
        print("*******")
        print("Moving file "+', '.join(vuln_files)+" from server to temporary/new folder")
        for file in vuln_files:
                os.system("mv "+file+" /Master/test_server_dir/files_from_step2/")
                print "* "+file+" is moved to /files_from_step2"
```

Listing 5.3: Code snippet of self-healing technique step 2.

Code snippet 5.3 demonstrates the use of step 2. It takes as input the vulnerable page which PHPIDS outputted, and the files which phpcs-security-audit detected contain vulnerabilities. Using the latter input, these files are simply moved from the web server to a separate folder of the server. Even though phpcs-security-audit presents more details on the vulnerabilities present in the files, these are not taken into consideration in this step.

Experiment 1 - Command injection

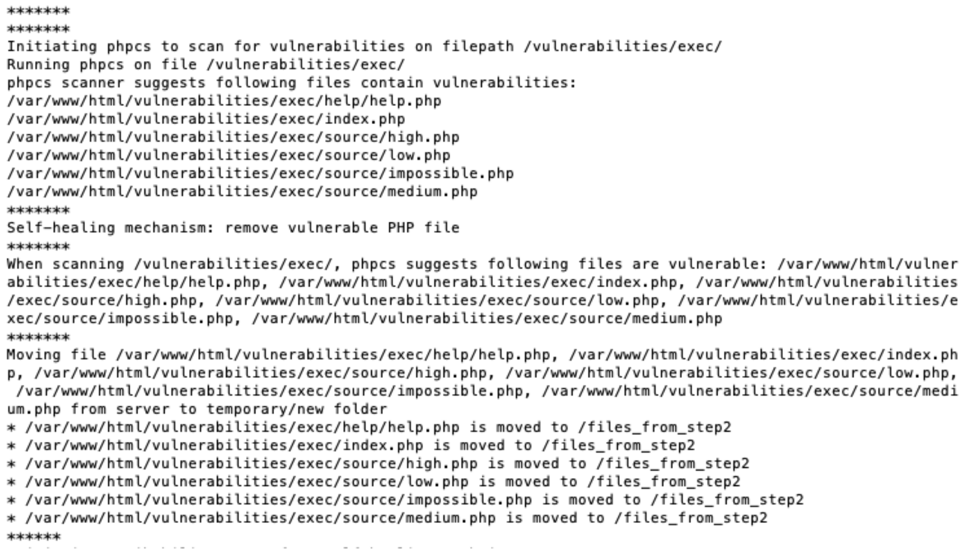
The same exploit and input parameters as in section 5.1 is used, namely **"google.com; ls ../../"**. The attack details are displayed in figure 5.6.



```
root@2828c1288deb:/Master/test_server_dir# python testing.py
************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-05T11:08:50+00:00', '10', '"dt id lfi"', '"REQUEST.ip=google.com%3B%20ls%20..%2
F..%2F POST.ip=google.com%3B%20ls%20..%2F..%2F"', '"%2Fvulnerabilities%2Fexec%2F"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-05T11:08:50+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.ip=google.com%3B%20ls%20..%2F..%2F POST.ip=google.com%3B%20ls%20..%2F..%2F"
Affected file/path: /vulnerabilities/exec/
Attack string: /vulnerabilities/exec/
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
two
*******
*******
Preparing availability criteria by checking availability of pages before initiating healing.
.......
http://localhost/vulnerabilities/javascript/help/help.php
500
http://localhost/vulnerabilities/javascript/index.php
302
200
http://localhost/vulnerabilities/javascript/source/medium.js
200
```

Figure 5.6: Command injection detected and step 2 is chosen as the self-healing technique in experiment 1 step 2.

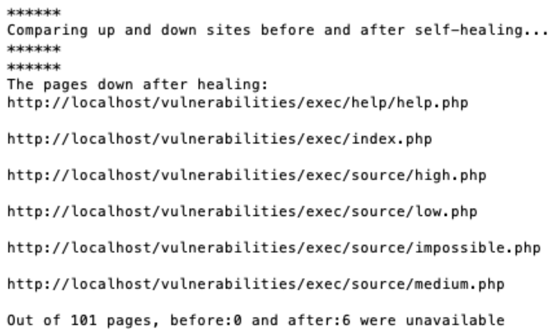
In contrast to the first experiment, phpcs-security-audit is launched (figure 5.7) since it is necessary to identify which files contain vulnerabilities. As illustrated in figure 5.7, the susceptible files are presented and moved to a confined directory, that is `/files_from_step2`, as part of the self-healing procedure.



```
*******
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/exec/
Running phpcs on file /vulnerabilities/exec/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/exec/help/help.php
/var/www/html/vulnerabilities/exec/index.php
/var/www/html/vulnerabilities/exec/source/high.php
/var/www/html/vulnerabilities/exec/source/low.php
/var/www/html/vulnerabilities/exec/source/impossible.php
/var/www/html/vulnerabilities/exec/source/medium.php
*******
Self-healing mechanism: remove vulnerable PHP file
*******
When scanning /vulnerabilities/exec/, phpcs suggests following files are vulnerable: /var/www/html/vulner
abilities/exec/help/help.php, /var/www/html/vulnerabilities/exec/index.php, /var/www/html/vulnerabilities
/exec/source/high.php, /var/www/html/vulnerabilities/exec/source/low.php, /var/www/html/vulnerabilities/e
xec/source/impossible.php, /var/www/html/vulnerabilities/exec/source/medium.php
*******
Moving file /var/www/html/vulnerabilities/exec/help/help.php, /var/www/html/vulnerabilities/exec/index.ph
p, /var/www/html/vulnerabilities/exec/source/high.php, /var/www/html/vulnerabilities/exec/source/low.php,
 /var/www/html/vulnerabilities/exec/source/impossible.php, /var/www/html/vulnerabilities/exec/source/medi
um.php from server to temporary/new folder
* /var/www/html/vulnerabilities/exec/help/help.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/exec/index.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/exec/source/high.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/exec/source/low.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/exec/source/impossible.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/exec/source/medium.php is moved to /files_from_step2
******
```

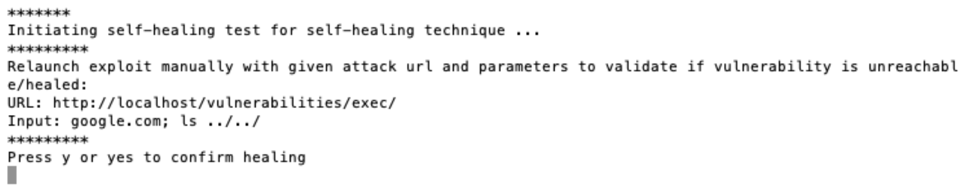
Figure 5.7: phpcs-security-audit scanning results and self-healing procedure of experiment 1 step 2.

The availability test results announces that six pages became unavailable after healing, pictured in figure 5.8. When relaunching the exploit to measure the extent of healing, it becomes apparent that the vulnerability is removed since the pages are inaccessible. It is not possible to interact with the pages containing vulnerabilities. Figure 5.10 displays the output in a web browser, and **"y"** is inputted in the script (figure 5.9) to confirm healing.



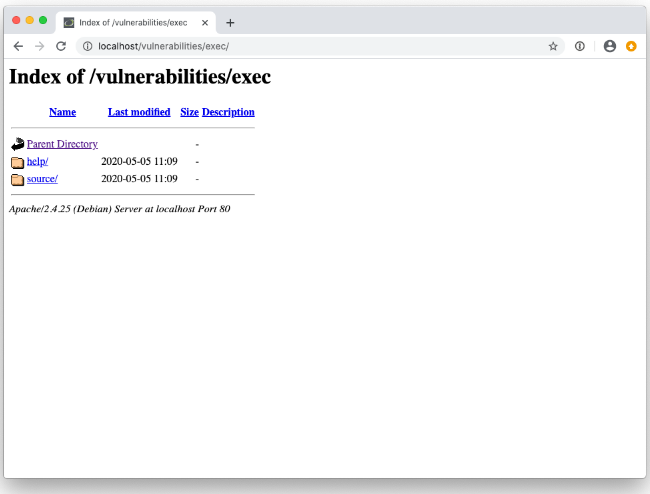
```
******
Comparing up and down sites before and after self-healing...
******
******
The pages down after healing:
http://localhost/vulnerabilities/exec/help/help.php

http://localhost/vulnerabilities/exec/index.php

http://localhost/vulnerabilities/exec/source/high.php

http://localhost/vulnerabilities/exec/source/low.php

http://localhost/vulnerabilities/exec/source/impossible.php

http://localhost/vulnerabilities/exec/source/medium.php

Out of 101 pages, before:0 and after:6 were unavailable
```

Figure 5.8: Availability test results in experiment 1 step 2.



```
*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameters to validate if vulnerability is unreachabl
e/healed:
URL: http://localhost/vulnerabilities/exec/
Input: google.com; ls ../../
*********
Press y or yes to confirm healing
█
```

Figure 5.9: Self-healing test in experiment 1 step 2.



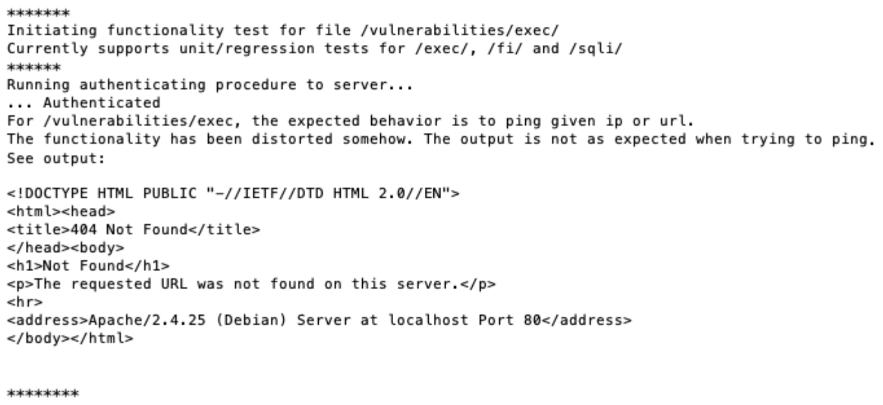
Index of /vulnerabilities/exec
localhost/vulnerabilities/exec/

Index of /vulnerabilities/exec

```
      Name            Last modified      Size  Description
-----------------------------------------------------------
 Parent Directory                          -
 help/              2020-05-05 11:09    -
 source/            2020-05-05 11:09    -
-----------------------------------------------------------
Apache/2.4.25 (Debian) Server at localhost Port 80
```

Figure 5.10: Browser display after executing self-healing approach experiment 1 step 2.

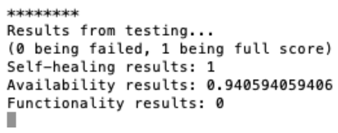
The functionality test results are presented in figure 5.11. It is not surprising that the test is unable to have the server ping a given URL, since the page executing the command is not available.



```
*******
Initiating functionality test for file /vulnerabilities/exec/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/exec, the expected behavior is to ping given ip or url.
The functionality has been distorted somehow. The output is not as expected when trying to ping.
See output:

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL was not found on this server.</p>
<hr>
<address>Apache/2.4.25 (Debian) Server at localhost Port 80</address>
</body></html>


********
```

Figure 5.11: Functionality test results in experiment 1 step 2.

Figure 5.12 displays the results from the criteria testing. Briefly, these results show that the healing of the vulnerability was a success, but the availability of the pages hosting these susceptibilities is not preserved. Hence, the functionality of the healed pages are also not maintained.

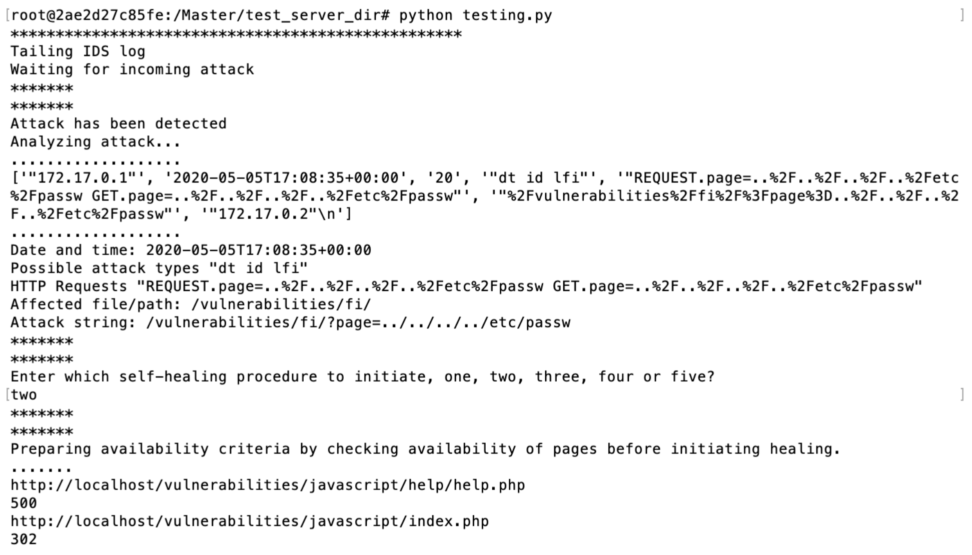


```
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 0.940594059406
Functionality results: 0
```

Figure 5.12: Results from the self-healing approach in experiment 1 step 2.

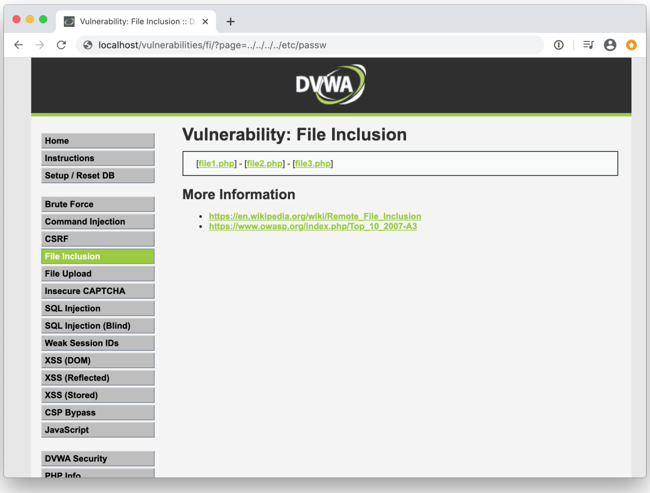
Experiment 2 - File inclusion

A file inclusion vulnerability allows for arbitrary files to be uploaded to the server. Displayed in figure 5.13 are the attack details with the malicious input "**../../../../etc/passw**" from the self-healing script. Figure 5.14 illustrates the attack scenario in the browser.



```
root@2ae2d27c85fe:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-05T17:08:35+00:00', '20', '"dt id lfi"', '"REQUEST.page=..%2F..%2F..%2F..%2Fetc
%2Fpasswd GET.page=..%2F..%2F..%2F..%2Fetc%2Fpasswd"', '"%2Fvulnerabilities%2Ffi%2F%3Fpage%3D..%2F..%2F..%2
F..%2Fetc%2Fpasswd"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-05T17:08:35+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.page=..%2F..%2F..%2F..%2Fetc%2Fpasswd GET.page=..%2F..%2F..%2F..%2Fetc%2Fpasswd"
Affected file/path: /vulnerabilities/fi/
Attack string: /vulnerabilities/fi/?page=../../../../etc/passwd
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
two
*******
*******
Preparing availability criteria by checking availability of pages before initiating healing.
.......
http://localhost/vulnerabilities/javascript/help/help.php
500
http://localhost/vulnerabilities/javascript/index.php
302
```

Figure 5.13: File inclusion detected and step 2 is chosen as the self-healing technique in experiment 2 step 2.



Vulnerability: File Inclusion :: D

localhost/vulnerabilities/fi/?page=../../../../etc/passwd

DVWA

Home
Instructions
Setup / Reset DB

Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript

DVWA Security
PHP Info

Vulnerability: File Inclusion

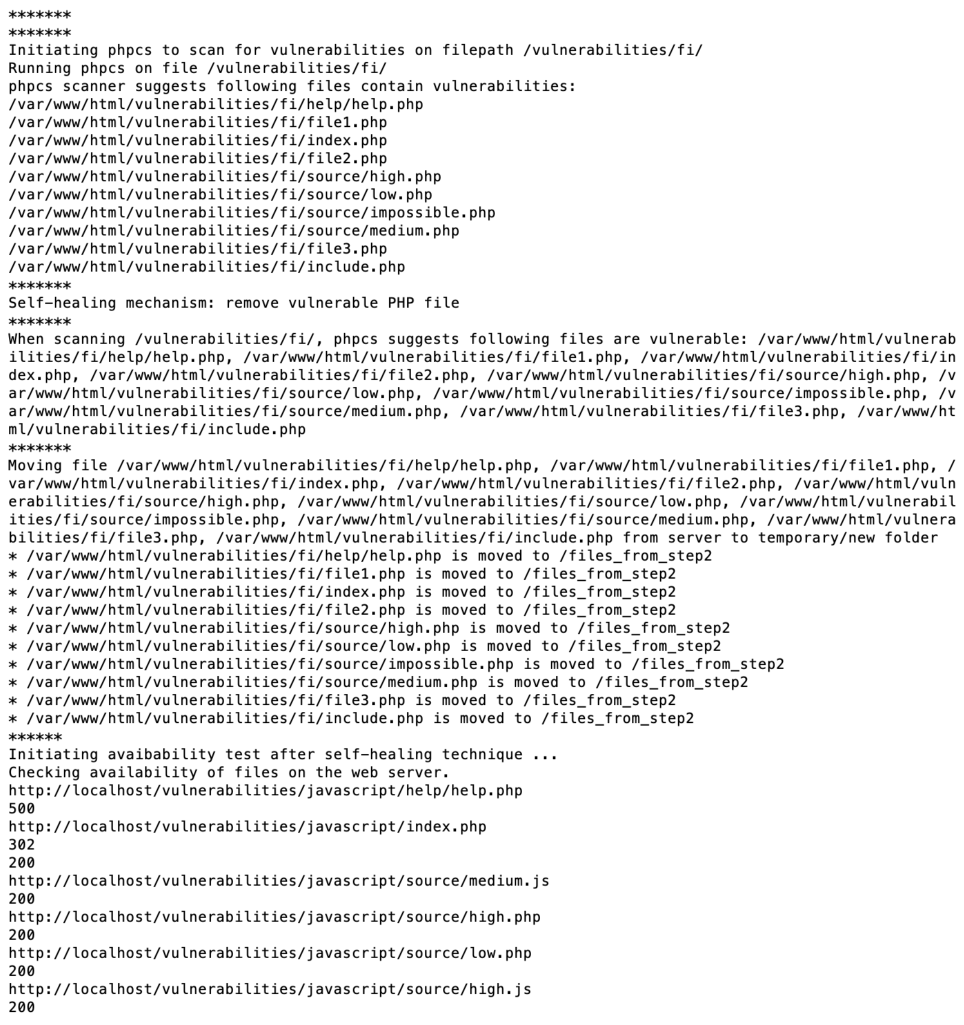
[file1.php] - [file2.php] - [file3.php]

More Information

- https://en.wikipedia.org/wiki/Remote_File_Inclusion
- https://www.owasp.org/index.php/Top_10_2007-A3

Figure 5.14: Browser with file inclusion attack string experiment 2 step 2.

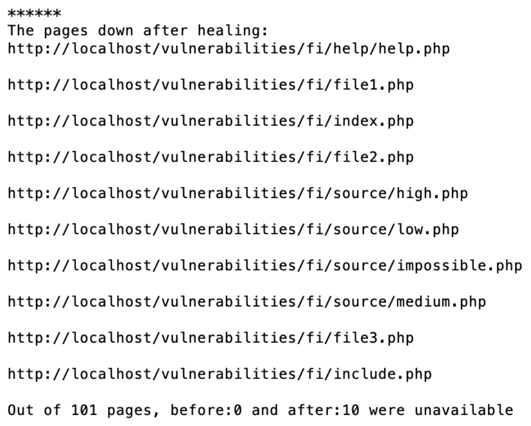
phpcs-security-audit is invoked, finding and presenting the suspected vulnerable files, and the self-healing procedure removes them from the server, demonstrated in figure 5.15.



```
*******
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/fi/
Running phpcs on file /vulnerabilities/fi/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/fi/help/help.php
/var/www/html/vulnerabilities/fi/file1.php
/var/www/html/vulnerabilities/fi/index.php
/var/www/html/vulnerabilities/fi/file2.php
/var/www/html/vulnerabilities/fi/source/high.php
/var/www/html/vulnerabilities/fi/source/low.php
/var/www/html/vulnerabilities/fi/source/impossible.php
/var/www/html/vulnerabilities/fi/source/medium.php
/var/www/html/vulnerabilities/fi/file3.php
/var/www/html/vulnerabilities/fi/include.php
*******
Self-healing mechanism: remove vulnerable PHP file
*******
When scanning /vulnerabilities/fi/, phpcs suggests following files are vulnerable: /var/www/html/vulnerab
ilities/fi/help/help.php, /var/www/html/vulnerabilities/fi/file1.php, /var/www/html/vulnerabilities/fi/in
dex.php, /var/www/html/vulnerabilities/fi/file2.php, /var/www/html/vulnerabilities/fi/source/high.php, /v
ar/www/html/vulnerabilities/fi/source/low.php, /var/www/html/vulnerabilities/fi/source/impossible.php, /v
ar/www/html/vulnerabilities/fi/source/medium.php, /var/www/html/vulnerabilities/fi/file3.php, /var/www/ht
ml/vulnerabilities/fi/include.php
*******
Moving file /var/www/html/vulnerabilities/fi/help/help.php, /var/www/html/vulnerabilities/fi/file1.php, /
var/www/html/vulnerabilities/fi/index.php, /var/www/html/vulnerabilities/fi/file2.php, /var/www/html/vuln
erabilities/fi/source/high.php, /var/www/html/vulnerabilities/fi/source/low.php, /var/www/html/vulnerabil
ities/fi/source/impossible.php, /var/www/html/vulnerabilities/fi/source/medium.php, /var/www/html/vulnera
bilities/fi/file3.php, /var/www/html/vulnerabilities/fi/include.php from server to temporary/new folder
* /var/www/html/vulnerabilities/fi/help/help.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/file1.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/index.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/file2.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/source/high.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/source/low.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/source/impossible.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/source/medium.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/file3.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/fi/include.php is moved to /files_from_step2
******
Initiating avaibability test after self-healing technique ...
Checking availability of files on the web server.
http://localhost/vulnerabilities/javascript/help/help.php
500
http://localhost/vulnerabilities/javascript/index.php
302
200
http://localhost/vulnerabilities/javascript/source/medium.js
200
http://localhost/vulnerabilities/javascript/source/high.php
200
http://localhost/vulnerabilities/javascript/source/low.php
200
http://localhost/vulnerabilities/javascript/source/high.js
200
```

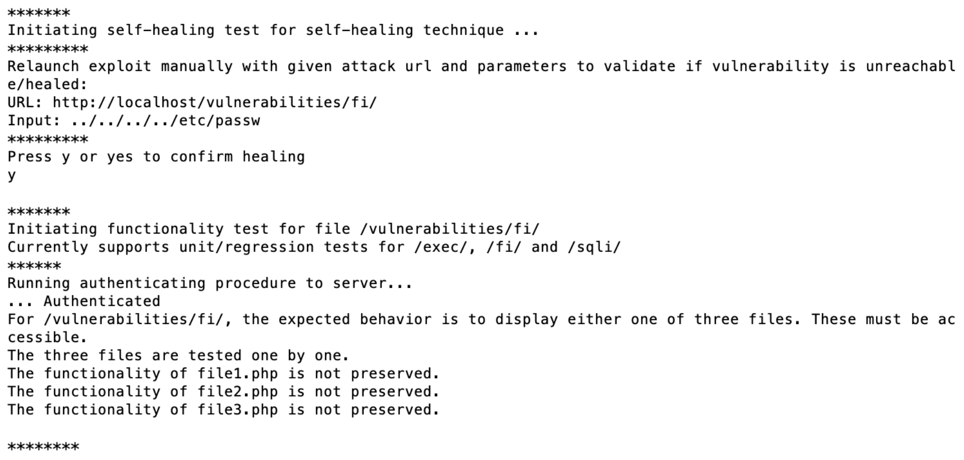
Figure 5.15: Results from phpcs-security-audit, the self-healing procedure is performed and the availability testing is initiated in experiment 2 step 2.

Figure 5.16 presents results from the availability test, and figure 5.17 from self-healing test and functionality test. As can be read from these results, there are expected down states from the files having been removed presented in the availability results. The functionality criteria is not met following the same reasoning as the previous test results; the pages are unavailable. However, the self-healing is seen as successful since the vulnerability is removed in such a way that the exploitable pieces are inaccessible. The scenario after healing is presented in a browser in figure 5.18.



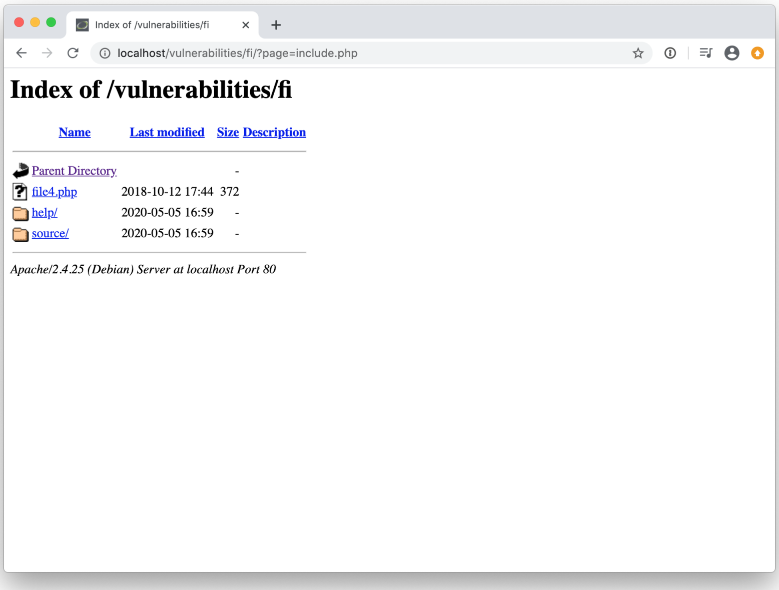
```
******
The pages down after healing:
http://localhost/vulnerabilities/fi/help/help.php

http://localhost/vulnerabilities/fi/file1.php

http://localhost/vulnerabilities/fi/index.php

http://localhost/vulnerabilities/fi/file2.php

http://localhost/vulnerabilities/fi/source/high.php

http://localhost/vulnerabilities/fi/source/low.php

http://localhost/vulnerabilities/fi/source/impossible.php

http://localhost/vulnerabilities/fi/source/medium.php

http://localhost/vulnerabilities/fi/file3.php

http://localhost/vulnerabilities/fi/include.php

Out of 101 pages, before:0 and after:10 were unavailable
```

Figure 5.16: Results from availability test when healing file inclusion in experiment 2 step 2.



```
*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameters to validate if vulnerability is unreachabl
e/healed:
URL: http://localhost/vulnerabilities/fi/
Input: ../../../../etc/passw
*********
Press y or yes to confirm healing
y

*******
Initiating functionality test for file /vulnerabilities/fi/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/fi/, the expected behavior is to display either one of three files. These must be ac
cessible.
The three files are tested one by one.
The functionality of file1.php is not preserved.
The functionality of file2.php is not preserved.
The functionality of file3.php is not preserved.

********
```

Figure 5.17: Results from self-healing test and functionality test when healing file inclusion in experiment 2 step 2.


Index of /vulnerabilities/fi
localhost/vulnerabilities/fi/?page=include.php

Index of /vulnerabilities/fi

Name	Last modified	Size	Description
Parent Directory		-	
file4.php	2018-10-12 17:44	372	
help/	2020-05-05 16:59	-	
source/	2020-05-05 16:59	-	

Apache/2.4.25 (Debian) Server at localhost Port 80

Figure 5.18: Web page presented in a browser after initiating self healing technique experiment 2 step 2.

Figure 5.19 summarized the results from each test in this experiment. The vulnerability is healed, and more than 90% of the pages are, according to the set criteria, available. However, the functionality of the given, vulnerable files are not preserved.

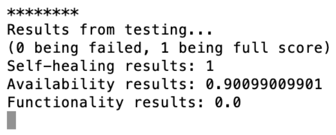
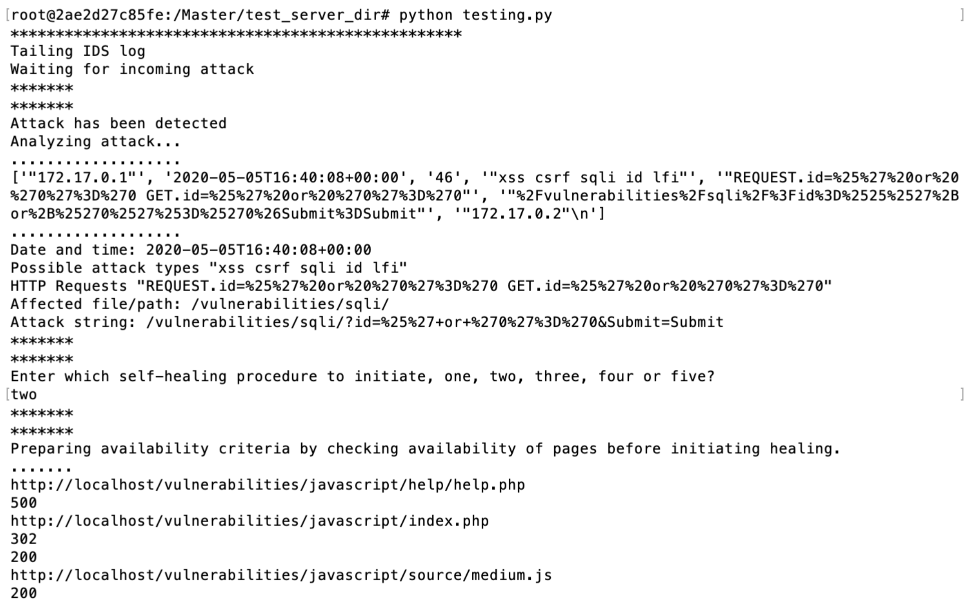

```
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 0.90099009901
Functionality results: 0.0
```

Figure 5.19: Results from testing in experiment 2 step 2.

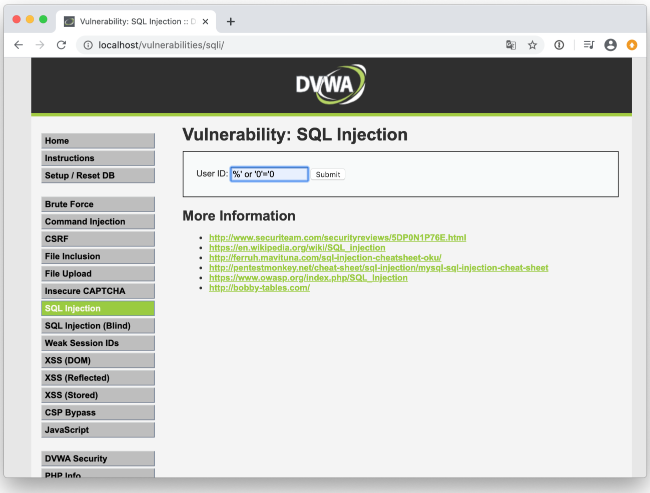
Experiment 3 - SQL injection

The current PHP file is vulnerable to SQL injections, and this is exploited using the attack string **"%' or '0'='0"** illustrated in figure 5.20 and 5.21.



```
root@2ae2d27c85fe:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-05T16:40:08+00:00', '46', '"xss csrf sqli id lfi"', '"REQUEST.id=%25%27%20or%20
%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"', '"%2Fvulnerabilities%2Fsqli%2F%3Fid%3D%2525%2527%2B
or%2B%25270%2527%253D%25270%26Submit%3DSubmit"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-05T16:40:08+00:00
Possible attack types "xss csrf sqli id lfi"
HTTP Requests "REQUEST.id=%25%27%20or%20%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"
Affected file/path: /vulnerabilities/sqli/
Attack string: /vulnerabilities/sqli/?id=%25%27+or+%270%27%3D%270&Submit=Submit
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
two
*******
*******
Preparing availability criteria by checking availability of pages before initiating healing.
.......
http://localhost/vulnerabilities/javascript/help/help.php
500
http://localhost/vulnerabilities/javascript/index.php
302
200
http://localhost/vulnerabilities/javascript/source/medium.js
200
```

Figure 5.20: SQL injection detected and step 2 is chosen as the self-healing technique in experiment 3 step 2.



Vulnerability: SQL Injection

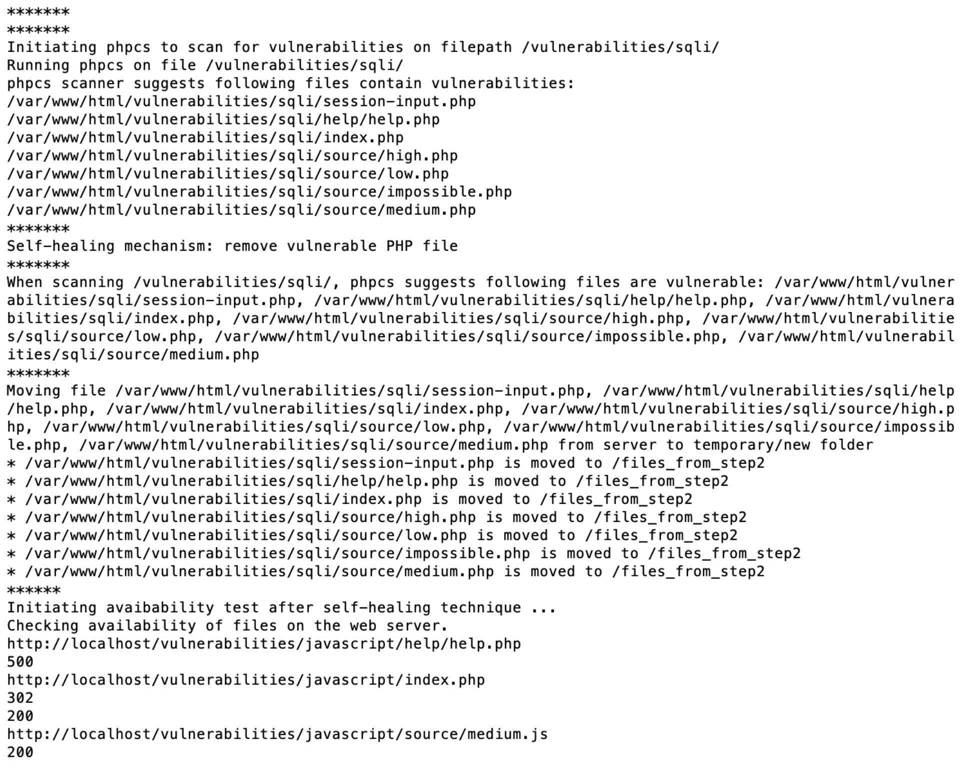
User ID: %' or '0'='0

More Information

- http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- https://en.wikipedia.org/wiki/SQL_injection
- http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
- https://www.owasp.org/index.php/SQL_Injection
- http://bobby-tables.com/

Figure 5.21: Browser with SQL injection attack string in experiment 3 step 2.

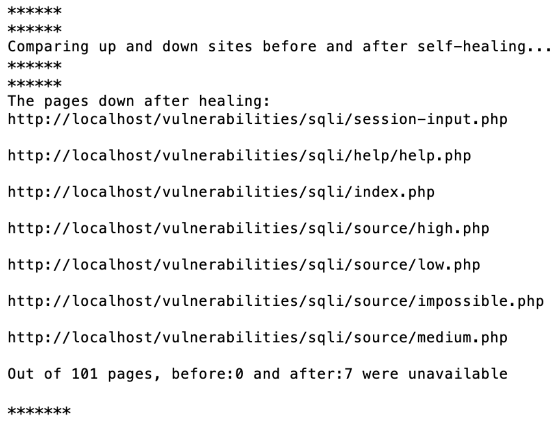
The results from phpcs-security-audit is presented in figure 5.22, as well as the self-healing process being executed. 7 pages are indicated to contain errors or vulnerabilities, and are therefore removed during healing.



```
*******
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/sqli/
Running phpcs on file /vulnerabilities/sqli/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/sqli/session-input.php
/var/www/html/vulnerabilities/sqli/help/help.php
/var/www/html/vulnerabilities/sqli/index.php
/var/www/html/vulnerabilities/sqli/source/high.php
/var/www/html/vulnerabilities/sqli/source/low.php
/var/www/html/vulnerabilities/sqli/source/impossible.php
/var/www/html/vulnerabilities/sqli/source/medium.php
*******
Self-healing mechanism: remove vulnerable PHP file
*******
When scanning /vulnerabilities/sqli/, phpcs suggests following files are vulnerable: /var/www/html/vulner
abilities/sqli/session-input.php, /var/www/html/vulnerabilities/sqli/help/help.php, /var/www/html/vulnera
bilities/sqli/index.php, /var/www/html/vulnerabilities/sqli/source/high.php, /var/www/html/vulnerabilitie
s/sqli/source/low.php, /var/www/html/vulnerabilities/sqli/source/impossible.php, /var/www/html/vulnerabil
ities/sqli/source/medium.php
*******
Moving file /var/www/html/vulnerabilities/sqli/session-input.php, /var/www/html/vulnerabilities/sqli/help
/help.php, /var/www/html/vulnerabilities/sqli/index.php, /var/www/html/vulnerabilities/sqli/source/high.p
hp, /var/www/html/vulnerabilities/sqli/source/low.php, /var/www/html/vulnerabilities/sqli/source/impossib
le.php, /var/www/html/vulnerabilities/sqli/source/medium.php from server to temporary/new folder
* /var/www/html/vulnerabilities/sqli/session-input.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/sqli/help/help.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/sqli/index.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/sqli/source/high.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/sqli/source/low.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/sqli/source/impossible.php is moved to /files_from_step2
* /var/www/html/vulnerabilities/sqli/source/medium.php is moved to /files_from_step2
******
Initiating avaibability test after self-healing technique ...
Checking availability of files on the web server.
http://localhost/vulnerabilities/javascript/help/help.php
500
http://localhost/vulnerabilities/javascript/index.php
302
200
http://localhost/vulnerabilities/javascript/source/medium.js
200
```

Figure 5.22: Results from phpcs-security-audit and the self-healing procedure is performed in experiment 3 step 2.

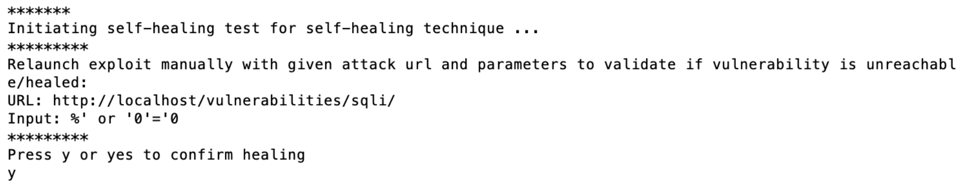
As anticipated, the files being removed in the self-healing procedure are the ones being inaccessible, presented in the availability test results shown in figure 5.23.



```
******
******
Comparing up and down sites before and after self-healing...
******
******
The pages down after healing:
http://localhost/vulnerabilities/sqli/session-input.php

http://localhost/vulnerabilities/sqli/help/help.php

http://localhost/vulnerabilities/sqli/index.php

http://localhost/vulnerabilities/sqli/source/high.php

http://localhost/vulnerabilities/sqli/source/low.php

http://localhost/vulnerabilities/sqli/source/impossible.php

http://localhost/vulnerabilities/sqli/source/medium.php

Out of 101 pages, before:0 and after:7 were unavailable

*******
```

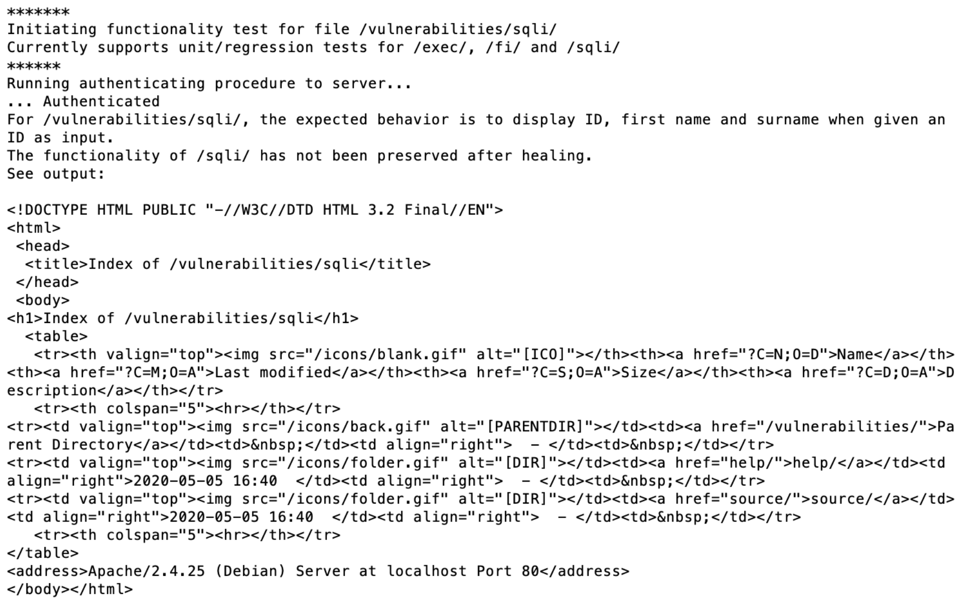
Figure 5.23: Availability test results from SQL injection experiment after healing in experiment 3 step 2.

The self-healing test gives the same result as the previous two experiments; the vulnerability is unreachable and, therefore, unexploitable. Therefore, **"y"** is inputted in the script to confirm healing, illustrated in figure 5.24. The functionality test results tell that the functionality is not preserved, clearly because the page holding the functionality is removed (figure 5.25).



```
*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameters to validate if vulnerability is unreachabl
e/healed:
URL: http://localhost/vulnerabilities/sqli/
Input: %' or '0'='0
*********
Press y or yes to confirm healing
y
```

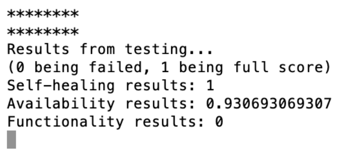
Figure 5.24: Results from self-healing test when healing SQL injection in experiment 3 step 2.



```
*******
Initiating functionality test for file /vulnerabilities/sqli/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/sqli/, the expected behavior is to display ID, first name and surname when given an
ID as input.
The functionality of /sqli/ has not been preserved after healing.
See output:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
 <head>
  <title>Index of /vulnerabilities/sqli</title>
 </head>
 <body>
<h1>Index of /vulnerabilities/sqli</h1>
  <table>
   <tr><th valign="top"><img src="/icons/blank.gif" alt="[ICO]"></th><th><a href="?C=N;O=D">Name</a></th>
<th><a href="?C=M;O=A">Last modified</a></th><th><a href="?C=S;O=A">Size</a></th><th><a href="?C=D;O=A">D
escription</a></th></tr>
   <tr><th colspan="5"><hr></th></tr>
<tr><td valign="top"><img src="/icons/back.gif" alt="[PARENTDIR]"></td><td><a href="/vulnerabilities/">Pa
rent Directory</a></td><td>&nbsp;</td><td align="right">  - </td><td>&nbsp;</td></tr>
<tr><td valign="top"><img src="/icons/folder.gif" alt="[DIR]"></td><td><a href="help/">help/</a></td><td
align="right">2020-05-05 16:40  </td><td align="right">  - </td><td>&nbsp;</td></tr>
<tr><td valign="top"><img src="/icons/folder.gif" alt="[DIR]"></td><td><a href="source/">source/</a></td>
<td align="right">2020-05-05 16:40  </td><td align="right">  - </td><td>&nbsp;</td></tr>
   <tr><th colspan="5"><hr></th></tr>
</table>
<address>Apache/2.4.25 (Debian) Server at localhost Port 80</address>
</body></html>
```

Figure 5.25: Results from functionality test in experiment 3 step 2.

Figure 5.26 lists the final results from testing in experiment 3 with the SQL injection. These show that the self-healing technique in this experiment was able to remove the detected vulnerability. However, both availability and functionality were sacrificed.



```
********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 0.930693069307
Functionality results: 0
```

Figure 5.26: Results from testing in experiment 3 step 2.

Summary of results

The self-healing process of removing susceptible files detected by the scanner is successful in these three experiments with regards to self-healing. The functionality results are also equal for the experiments, being that the functionality of the pages are absent in consequence of removing them. There are variations in the availability results ranging from approximately 0.90 to 0.94. However, for each experiment, all the files being removed are the ones being detected by the vulnerability scanner. Other files on the server are not affected by their disappearance. From this perspective, they all score equally on the availability scale. The results are summarized in table 5.2.

	Self-healing test	**Availability test**	**Functionality test**	**Average**
Exp. 1	1	0.94	0	0.647
Exp. 2	1	0.90	0	0.633
Exp. 3	1	0.93	0	0.643
Average	1	0.923	0	0.641

Table 5.2: Summary of results for step 2. The first three rows give the results from each experiments, including the average performance of each experiment in the final column. The last row gives the average results from all experiments, and the bottom right cell gives an idea of how well the self-healing technique performed concerning the evaluation criteria. Each experiment achieves similar results, where the only diversity is caused by the availability results. Even though the functionality was not kept (**0** as the result) in either experiment, these results were expected and necessary for the self-healing to be a success. However, it is preferable to keep the functionality after healing.

5.3 Step 3 - Remove susceptible line of code in PHP file

Based on the results from the vulnerability scanner, the self-healing technique removes the lines of code which are indicated to contain vulnerabilities. There was conducted three experiments, trying to heal a SQL injection, file inclusion and command injection vulnerability.

```python
def step3(filepath, loc):
        #filepath: path from PHPIDS
        #loc: line(s) of code in filepath containing susceptibilities
        print("*******")
        print("*******")
        print("Self-healing mechanism: remove vulnerable line of code")
        print("*******")
        f = open(filepath, "r")
        lines = f.readlines()
        f.close()
        print("Creating .old file for file "+filepath)
        name = filepath[len("/var/www/html/vulnerabilities/"):]
        name = name.replace("/","_")
        os.system("mv "+filepath+" /Master/test_server_dir/
    files_from_step3/"+name+".old")
```

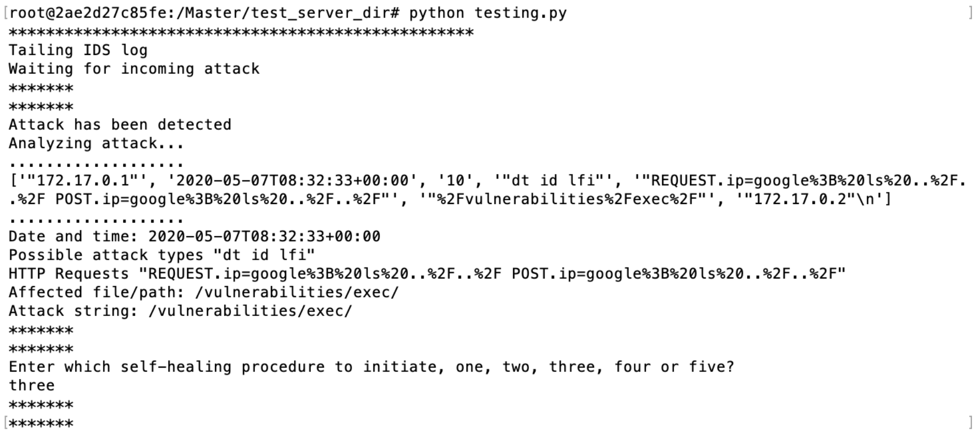
```
        print("Copying lines of code to old file to path oldfiles/")
        mod_lines = lines
        print("Modifying original file's vulnerable lines of code")
        for l in loc:
                mod_line = "//"+lines[l-1]
                mod_lines[l-1] = mod_line
        print "The lines with modifications are: \n"
        for mline in mod_lines:
                print mline
        print("Overwriting original file with new, modified lines")
        nw = open(filepath, "w")
        nw.writelines(mod_lines)
        nw.close()
        print "*******"
        print "*******"
```

Listing 5.4: Code snippet of self-healing technique step 3.

In code snippet 5.4, the functionality of step 3 is presented. In order to "remove" or isolate the vulnerable lines of code, the solution comments out ("//" in PHP) the lines according to the list of vulnerable code locations from the input. It reads the file in question, moves the original file to a separate folder and renames it to `name.old`, and comments out the vulnerable lines of code in the vulnerable file. These modifications are written to the original file on the web server.

Experiment 1 - Command injection

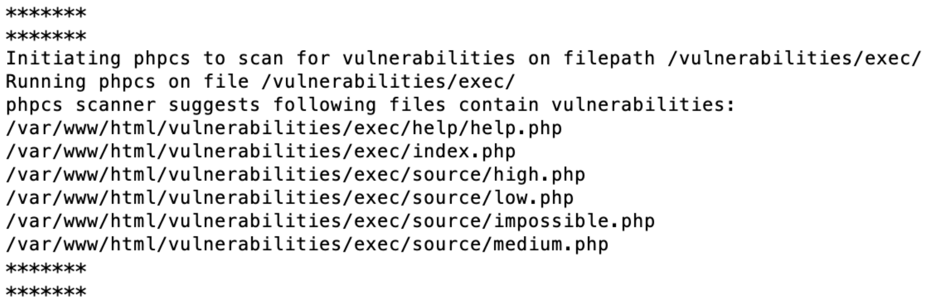
The same exploit and input parameters as in section 5.1 are used, namely **"google.com; ls ../../"**. The attack details are displayed in figure 5.27.



```
root@2ae2d27c85fe:/Master/test_server_dir# python testing.py
************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
..................
['"172.17.0.1"', '2020-05-07T08:32:33+00:00', '10', '"dt id lfi"', '"REQUEST.ip=google%3B%20ls%20..%2F.
.%2F POST.ip=google%3B%20ls%20..%2F..%2F"', '"%2Fvulnerabilities%2Fexec%2F"', '"172.17.0.2"\n']
..................
Date and time: 2020-05-07T08:32:33+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.ip=google%3B%20ls%20..%2F..%2F POST.ip=google%3B%20ls%20..%2F..%2F"
Affected file/path: /vulnerabilities/exec/
Attack string: /vulnerabilities/exec/
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
three
*******
*******
```

Figure 5.27: Command injection detected and step 3 is chosen as the self-healing technique in experiment 1 step 3.

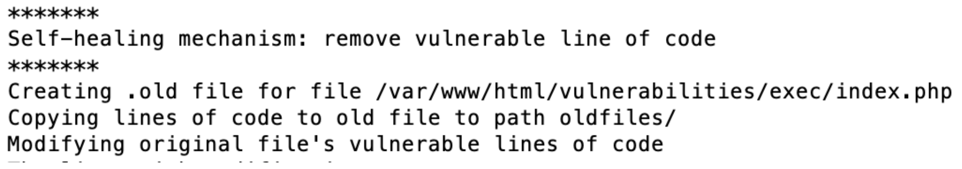
As in the previous experiments, phpcs-security-audit uses the attacked filepath identified by PHPIDS when scanning. The results are presented and, as expected, equal to earlier scans of the same path, stated in figure 5.28.



```
*******
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/exec/
Running phpcs on file /vulnerabilities/exec/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/exec/help/help.php
/var/www/html/vulnerabilities/exec/index.php
/var/www/html/vulnerabilities/exec/source/high.php
/var/www/html/vulnerabilities/exec/source/low.php
/var/www/html/vulnerabilities/exec/source/impossible.php
/var/www/html/vulnerabilities/exec/source/medium.php
*******
*******
```

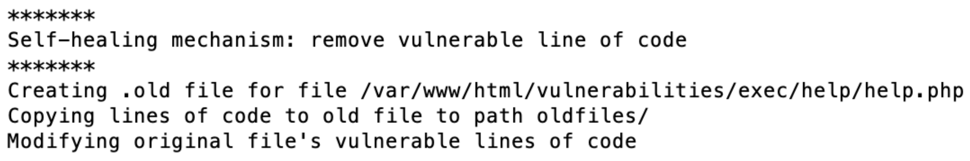
Figure 5.28: phpcs-security-audit results when scanning the identified, attacked filepath in experiment 1 step 3.

For each vulnerable file identified by the scanner, their corresponding vulnerable locations are commented out from the code. Example outputs from the self-healing script for two of the vulnerable files are shown in figure 5.29 and 5.30.



```
*******
Self-healing mechanism: remove vulnerable line of code
*******
Creating .old file for file /var/www/html/vulnerabilities/exec/index.php
Copying lines of code to old file to path oldfiles/
Modifying original file's vulnerable lines of code
```

Figure 5.29: First example output for self-healing script when healing vulnerabilities in experiment 1 in step 3.



```
*******
Self-healing mechanism: remove vulnerable line of code
*******
Creating .old file for file /var/www/html/vulnerabilities/exec/help/help.php
Copying lines of code to old file to path oldfiles/
Modifying original file's vulnerable lines of code
```

Figure 5.30: Second example output for self-healing script when healing vulnerabilities in experiment 1 in step 3.

One example of healed files from the healing process is file `low.php`, as identified by phpcs-security-audit (figure 5.28). Inspecting the scanner results from this file, illustrated in listing 5.5 from appendix A.2, the lines being commented out is demonstrated. Listing 5.6 shows how the PHP file appears after healing; the vulnerable lines of code identified by the scanner (3, 5, 10 and 14) are neutralized in the script.

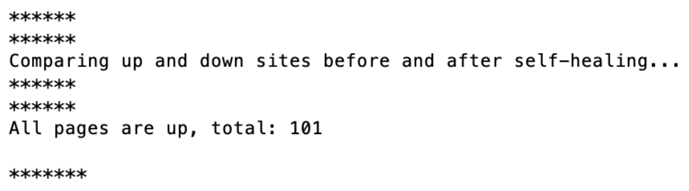
```
FILE: /var/www/html/vulnerabilities/exec/source/low.php
----------------------------------------------------------------------

FOUND 0 ERRORS AND 4 WARNINGS AFFECTING 4 LINES
----------------------------------------------------------------------
  3 | WARNING | User input detetected with $_POST.
  5 | WARNING | User input detetected with $_REQUEST.
 10 | WARNING | System program execution function shell_exec() detected
    with dynamic parameter
 14 | WARNING | System program execution function shell_exec() detected
    with dynamic parameter
----------------------------------------------------------------------
```

Listing 5.5: Parts of phpcs-security-audit scanning results of `low.php` in experiment 1 step 3.

```php
<?php

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

  // Determine OS and execute the ping command.
  if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
    // Windows
//    $cmd = shell_exec( 'ping  ' . $target );
  }
  else {
    // *nix
//    $cmd = shell_exec( 'ping  -c 4 ' . $target );
  }

  // Feedback for the end user
  $html .= "<pre>{$cmd}</pre>";
}

?>
```

Listing 5.6: `low.php` after healing in experiment 1 step 3

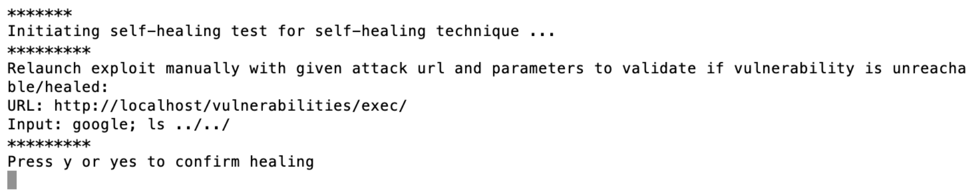
The availability test results is displayed in figure 5.31. These results state that all pages are up and existing, namely, no `curl` query return "404" in the HTTP response.



```
******
******
Comparing up and down sites before and after self-healing...
******
******
All pages are up, total: 101

*******
```

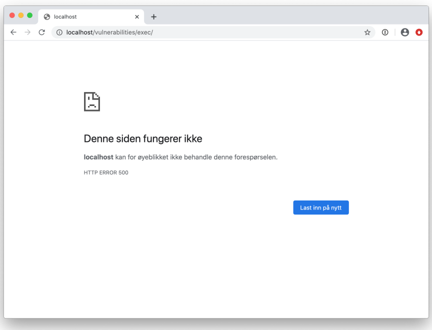
Figure 5.31: Results from availability test in experiment 1 in step 3.

The self-healing test scenario is expressed in figure 5.32, 5.33 and 5.34. These results show that the vulnerability is healed since the vulnerable code is not present on the sites. They also show that the sites are available. However, it seems the healing approach, commenting out vulnerable lines of code, have caused errors in the server code, thus "HTTP ERROR 500" error code.



```
*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameters to validate if vulnerability is unreacha
ble/healed:
URL: http://localhost/vulnerabilities/exec/
Input: google; ls ../../
*********
Press y or yes to confirm healing
```

Figure 5.32: Self-healing test in experiment 1 in step 3.

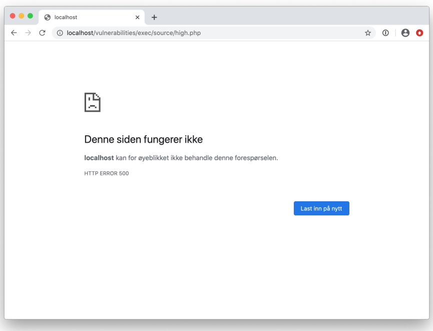


Denne siden fungerer ikke

localhost kan for øyeblikket ikke behandle denne forespørselen.

HTTP ERROR 500

Figure 5.33: Browser showing internal error experiment 1 in step 3.



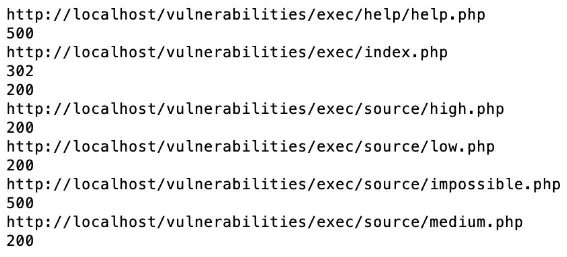
Denne siden fungerer ikke

localhost kan for øyeblikket ikke behandle denne forespørselen.

HTTP ERROR 500

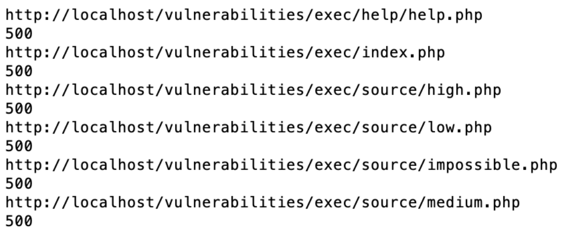
Figure 5.34: Browser showing internal error for page `high.php` in experiment 1 in step 3.

Looking at the results from the availability tests before (figure 5.35) and after (figure 5.36), we can conclude that there appeared internal server issues from the healing procedure which need to be resolved to achieve expected functionality.



```
http://localhost/vulnerabilities/exec/help/help.php
500
http://localhost/vulnerabilities/exec/index.php
302
200
http://localhost/vulnerabilities/exec/source/high.php
200
http://localhost/vulnerabilities/exec/source/low.php
200
http://localhost/vulnerabilities/exec/source/impossible.php
500
http://localhost/vulnerabilities/exec/source/medium.php
200
```

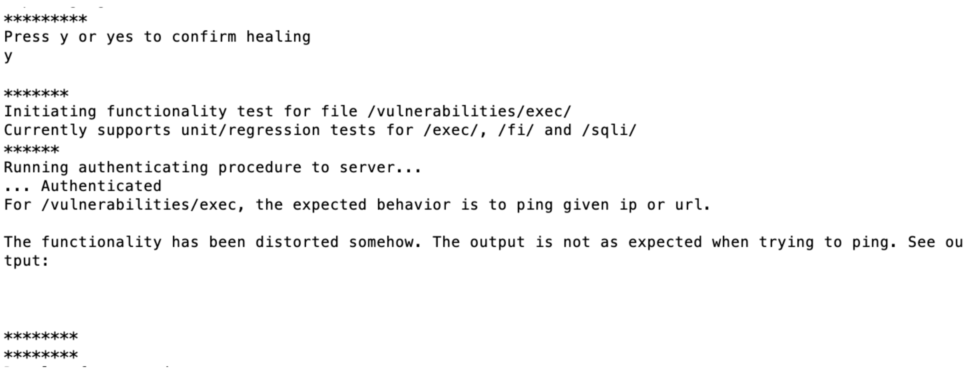
Figure 5.35: Availability results before healing in experiment 1 step 3.



```
http://localhost/vulnerabilities/exec/help/help.php
500
http://localhost/vulnerabilities/exec/index.php
500
http://localhost/vulnerabilities/exec/source/high.php
500
http://localhost/vulnerabilities/exec/source/low.php
500
http://localhost/vulnerabilities/exec/source/impossible.php
500
http://localhost/vulnerabilities/exec/source/medium.php
500
```

Figure 5.36: Availability result after healing in experiment 1 step 3.

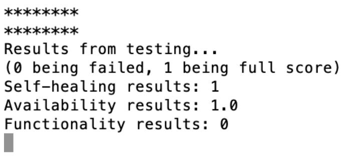
It is therefore not surprising that the functionality test fails, since there are errors on the server making it not perform correctly. The details from the test are demonstrated in figure 5.37.



```
*********
Press y or yes to confirm healing
y

*******
Initiating functionality test for file /vulnerabilities/exec/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/exec, the expected behavior is to ping given ip or url.

The functionality has been distorted somehow. The output is not as expected when trying to ping. See ou
tput:



********
********
```

Figure 5.37: Functionality results in experiment 1 step 3.

The results from the experiment is presented in figure 5.38, showing that the self-healing and availability criteria are met, while the functionality criteria is not.

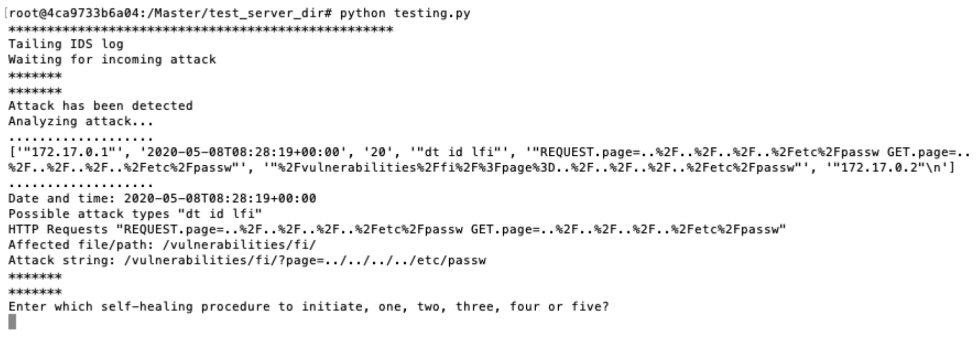


```
********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: 0
```

Figure 5.38: Results from experiment 1 step 3.

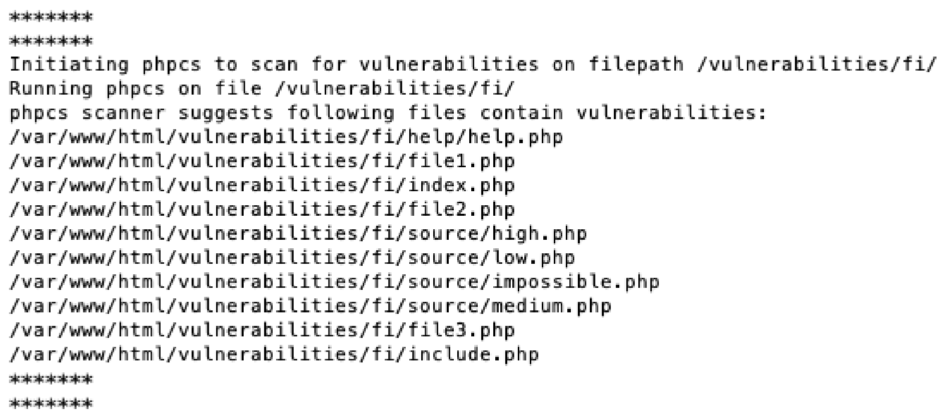
Experiment 2 - File inclusion

The experiment follows the same setup as the previous experiments with file inclusions, namely input **"../../../../etc/passw"**, with details in figure 5.39.



```
[root@4ca9733b6a04:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-08T08:28:19+00:00', '20', '"dt id lfi"', '"REQUEST.page=..%2F..%2F..%2F..%2Fetc%2Fpassw GET.page=..
%2F..%2F..%2F..%2Fetc%2Fpassw"', '"%2Fvulnerabilities%2Ffi%2F%3Fpage%3D..%2F..%2F..%2F..%2Fetc%2Fpassw"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-08T08:28:19+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.page=..%2F..%2F..%2F..%2Fetc%2Fpassw GET.page=..%2F..%2F..%2F..%2Fetc%2Fpassw"
Affected file/path: /vulnerabilities/fi/
Attack string: /vulnerabilities/fi/?page=../../../../etc/passw
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
█
```

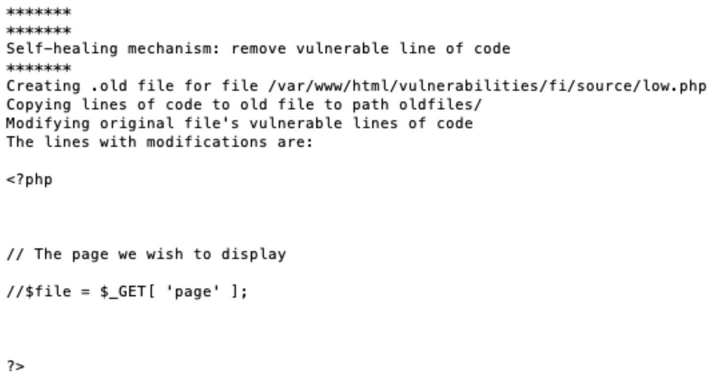
Figure 5.39: File inclusion attempt detected and step 3 is chosen as the self-healing technique in experiment 1 step 3.



```
*******
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/fi/
Running phpcs on file /vulnerabilities/fi/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/fi/help/help.php
/var/www/html/vulnerabilities/fi/file1.php
/var/www/html/vulnerabilities/fi/index.php
/var/www/html/vulnerabilities/fi/file2.php
/var/www/html/vulnerabilities/fi/source/high.php
/var/www/html/vulnerabilities/fi/source/low.php
/var/www/html/vulnerabilities/fi/source/impossible.php
/var/www/html/vulnerabilities/fi/source/medium.php
/var/www/html/vulnerabilities/fi/file3.php
/var/www/html/vulnerabilities/fi/include.php
*******
*******
```

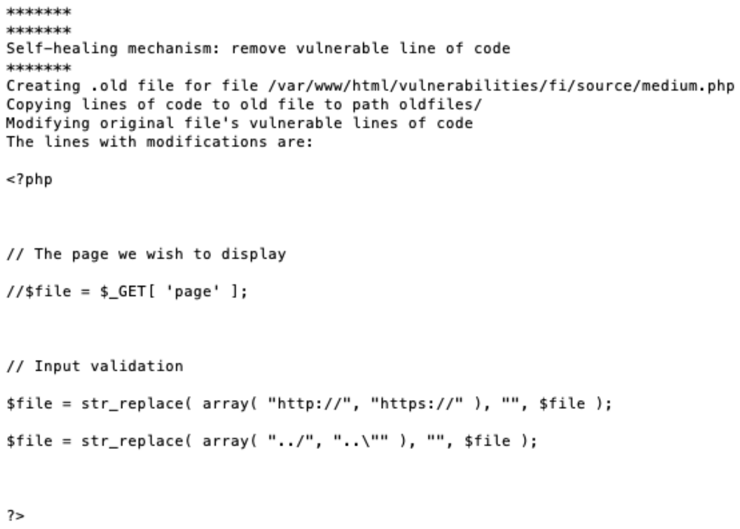
Figure 5.40: phpcs-security-audit results when scanning the identified, attacked filepath in experiment 2 step 3.

phpcs-security audit outputs which files are vulnerable, shown in figure 5.40. Using these results from the scanner, the self-healing procedure initiates healing of all files registered as vulnerable. In the two examples, figure 5.41 and 5.42, respectively files `low.php` and `medium.php` are shown with modifications applied. In both cases, the line of code in which gets the requested page is commented out. The error message of phpcs-security-audit's scan is in listing 5.7, implying that there is an issue using the function `$_GET` with user input.



```
*******
*******
Self-healing mechanism: remove vulnerable line of code
*******
Creating .old file for file /var/www/html/vulnerabilities/fi/source/low.php
Copying lines of code to old file to path oldfiles/
Modifying original file's vulnerable lines of code
The lines with modifications are:

<?php


// The page we wish to display

//$file = $_GET[ 'page' ];


?>
```

Figure 5.41: File `low.php` healed in experiment 2 step 3.



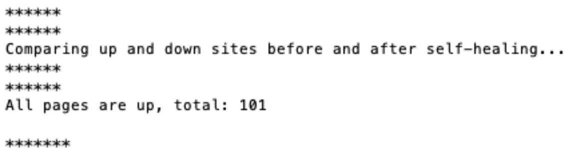
```
*******
*******
Self-healing mechanism: remove vulnerable line of code
*******
Creating .old file for file /var/www/html/vulnerabilities/fi/source/medium.php
Copying lines of code to old file to path oldfiles/
Modifying original file's vulnerable lines of code
The lines with modifications are:

<?php


// The page we wish to display

//$file = $_GET[ 'page' ];


// Input validation

$file = str_replace( array( "http://", "https://" ), "", $file );

$file = str_replace( array( "../", "..\"" ), "", $file );


?>
```

Figure 5.42: File `medium.php` healed in experiment 2 step 3.

```
 4 | WARNING | User input detetected with $_GET.
```

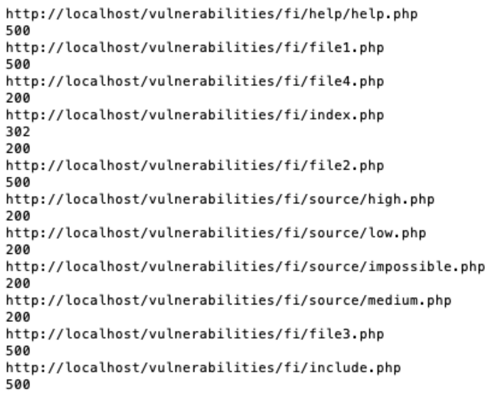
Listing 5.7: Parts of phpcs-security-audit scanning results of `low.php` and `medium.php` in experiment 2 step 3.

The availability results are equal to the previous experiment, i.e., all pages are up. This can be confirmed when using a browser to try and access the pages. However, the HTTP responses only changed from "200" to "500" for one file, namely `index.php`. The details for the availability testing are displayed in figure 5.43, 5.44 and 5.45.



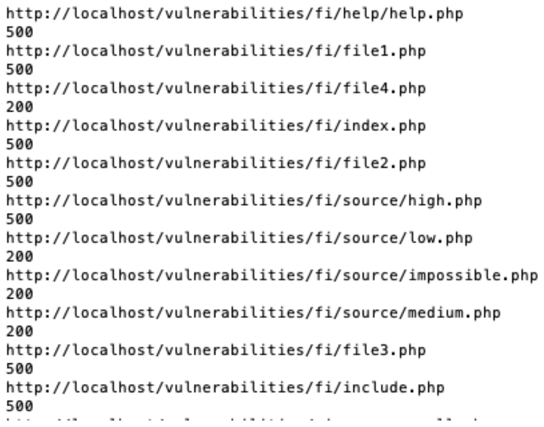
```
******
******
Comparing up and down sites before and after self-healing...
******
******
All pages are up, total: 101

*******
```

Figure 5.43: Results from availability test in experiment 2 step 3.



```
http://localhost/vulnerabilities/fi/help/help.php
500
http://localhost/vulnerabilities/fi/file1.php
500
http://localhost/vulnerabilities/fi/file4.php
200
http://localhost/vulnerabilities/fi/index.php
302
200
http://localhost/vulnerabilities/fi/file2.php
500
http://localhost/vulnerabilities/fi/source/high.php
200
http://localhost/vulnerabilities/fi/source/low.php
200
http://localhost/vulnerabilities/fi/source/impossible.php
200
http://localhost/vulnerabilities/fi/source/medium.php
200
http://localhost/vulnerabilities/fi/file3.php
500
http://localhost/vulnerabilities/fi/include.php
500
```

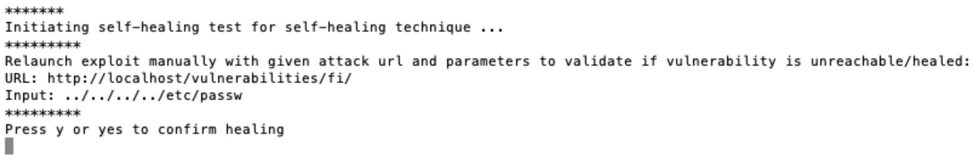
Figure 5.44: Availability results before healing in experiment 2 step 3.



```
http://localhost/vulnerabilities/fi/help/help.php
500
http://localhost/vulnerabilities/fi/file1.php
500
http://localhost/vulnerabilities/fi/file4.php
200
http://localhost/vulnerabilities/fi/index.php
500
http://localhost/vulnerabilities/fi/file2.php
500
http://localhost/vulnerabilities/fi/source/high.php
500
http://localhost/vulnerabilities/fi/source/low.php
200
http://localhost/vulnerabilities/fi/source/impossible.php
200
http://localhost/vulnerabilities/fi/source/medium.php
200
http://localhost/vulnerabilities/fi/file3.php
500
http://localhost/vulnerabilities/fi/include.php
500
```

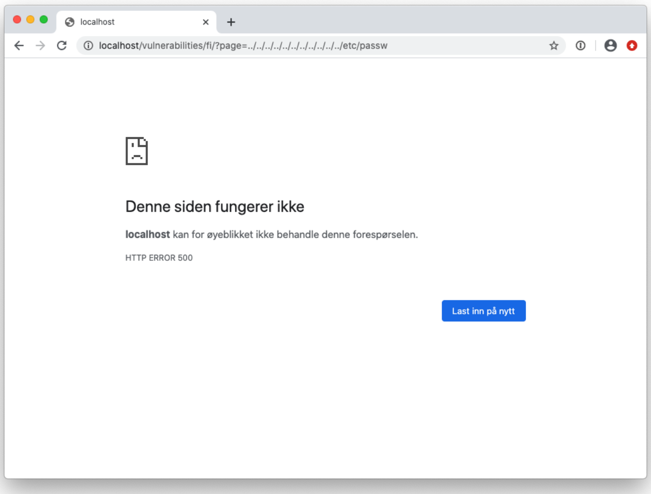
Figure 5.45: Availability result after healing in experiment 2 step 3.

The self-healing test details are given in the script, as shown in figure 5.46. When trying to execute a file inclusion attempt towards the page in a browser (figure 5.47), it is clear that the vulnerability is not present. Internal server errors are preventing normal execution of the page.



```
*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameters to validate if vulnerability is unreachable/healed:
URL: http://localhost/vulnerabilities/fi/
Input: ../../../../etc/passw
*********
Press y or yes to confirm healing
█
```

Figure 5.46: Self-healing test in experiment 2 step 3.



localhost

localhost/vulnerabilities/fi/?page=../../../../../../../../../../../etc/passw

Denne siden fungerer ikke

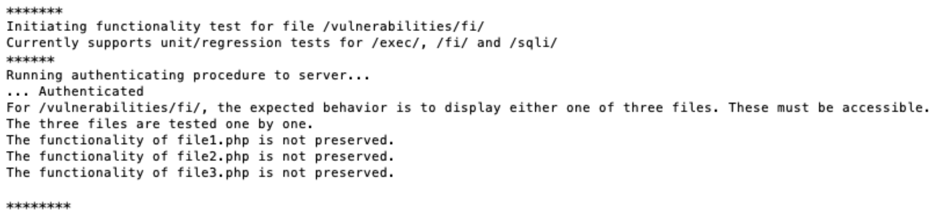
localhost kan for øyeblikket ikke behandle denne forespørselen.

HTTP ERROR 500

Last inn på nytt

Figure 5.47: Testing if the file inclusion susceptibility is removed after healing in experiment 2 in step 3.

As with experiment 1, the functionality of the page is distorted. This is stated in the self-healing script presented in figure 5.48, but can also be seen when running manual tests. Figure 5.49 shows how the page is reachable, but the internal server errors clutters its functionality.



```
*******
Initiating functionality test for file /vulnerabilities/fi/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/fi/, the expected behavior is to display either one of three files. These must be accessible.
The three files are tested one by one.
The functionality of file1.php is not preserved.
The functionality of file2.php is not preserved.
The functionality of file3.php is not preserved.

********
```

Figure 5.48: Functionality test results for experiment 2 step 3.

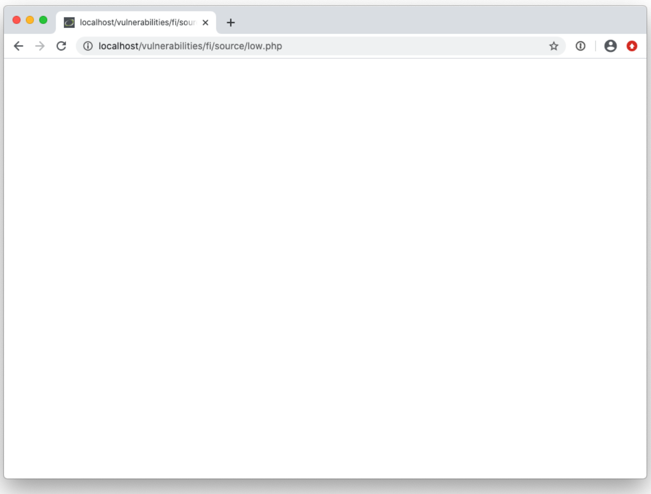
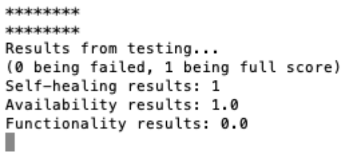

localhost/vulnerabilities/fi/sour
localhost/vulnerabilities/fi/source/low.php

Figure 5.49: Example of reachable page with distorted functionality in experiment 2 step 3.

The results after all the tests are presented in figure 5.50. Experiment 2 achieves the same results as experiment 1 with regards to the evaluation criteria measurements.

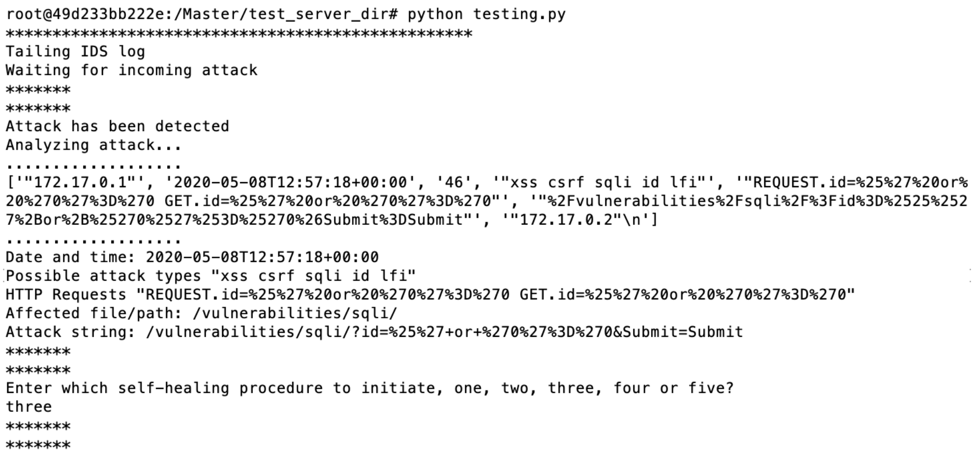


```
********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: 0.0
```

Figure 5.50: Results after testing in experiment 2 step 3.

Experiment 3 - SQL injection

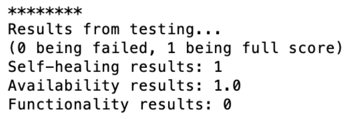
Using attack string **"%' or '0'='0"** as in previous SQL injection experiments, the self-healing script output is displayed in figure 5.51.



```
root@49d233bb222e:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-08T12:57:18+00:00', '46', '"xss csrf sqli id lfi"', '"REQUEST.id=%25%27%20or%
20%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"', '"%2Fvulnerabilities%2Fsqli%2F%3Fid%3D%2525%252
7%2Bor%2B%25270%2527%253D%25270%26Submit%3DSubmit"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-08T12:57:18+00:00
Possible attack types "xss csrf sqli id lfi"
HTTP Requests "REQUEST.id=%25%27%20or%20%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"
Affected file/path: /vulnerabilities/sqli/
Attack string: /vulnerabilities/sqli/?id=%25%27+or+%270%27%3D%270&Submit=Submit
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
three
*******
*******
```

Figure 5.51: SQL injection detected and step 3 is chosen as the self-healing technique.

For the rest of the execution of the program, the results are similar to the previous two experiments. The server is experiencing internal errors due to the healing technique, but the pages are available (HTTP error response is not 404). The vulnerability of SQL injection is removed, however. The results are summarized in figure 5.52.



```
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: 0
```

Figure 5.52: Results from experiment 3 step 3.

Summary of results

In this section, the self-healing technique of removing vulnerable lines of code in PHP files have been tested. Based on three experiments, it is possible to conclude that the vulnerabilities are not reachable. Hence, the server is healed. All experiments also resulted in the pages being available and "up". However, the test for availability did not include unavailability due to internal server errors, such as "500 Internal Server Error". As expected, when removing arbitrary lines in the PHP code, this can cause syntax errors and, therefore, server errors. For that reason, the functionality was not kept in any of the experiments - the server never worked as intended. The self-healing technique is able to heal the vulnerability without compromising uptime. Still, the functionality of the healed paged has been reduced, and similar results will probably appear if tested with other vulnerable PHP pages. The results from the experiments are summarized in table 5.3.

	Self-healing test	**Availability test**	**Functionality test**	**Average**
Exp. 1	1	1	0	0.666
Exp. 2	1	1	0	0.666
Exp. 3	1	1	0	0.666
Average	1	1	0	0.666

Table 5.3: Summary of results for step 3. The first three rows give the results from each experiments, including the average performance of each experiment in the final column. The last row gives the average results from all experiments, and the bottom right cell gives an idea of how well the self-healing technique performed concerning the evaluation criteria. As this table illustrates, each experiment has the same result for each performance test. Even though the functionality was not kept (**0** as the result) in either experiment, these results were expected and necessary for the self-healing to be a success. However, it is preferable to keep the functionality after healing.

5.4 Step 4 - Add sanitization to input

In this self-healing technique, basic sanitization and filters are applied to user input. The type of vulnerability detected decides what kind of sanitization is used. The self-healing procedure currently supports XSS, SQL injections, directory traversal exploits, and command injections by interpreting PHPIDS and phpcs-security-audit output. The attacks not falling into either three categorizations are sanitized uniformly using a simple, allround method.

First scenario: XSS

if-condition to trigger scenario: `("User input" and postget) in '\t'.join(phpcs_error_messages) or "HTML construction with direct user input" in '\t'.join(phpcs_error_messages) and "xss" in phpids_type`
Filter or sanitization method used: `htmlspecialchars()`
When the detected breach is categorized as an XSS attack, the healing is performed by

sanitizing the user input using `htmlspecialchars()`. The PHP function is popular for mitigating script injections since it converts special characters to HTML entities[1].

Second scenario: SQL injection

`if`-condition to trigger scenario: `"sqli" in phpids_type and ("mysqli_query" and "dynamic parameter") in '\t'.join(phpcs_error_messages)`
Filter or sanitization method used: `mysqli_real_escape_string()`
If the attack string is detected to be an SQL injection, the function `mysqli_real_escape_string()` is applied in the current PHP file. It escapes special characters in a string for use in an SQL statement[2].

Third scenario: Directory traversal attacks and command injection

`if`-condition to trigger scenario: `("dt" in phpids_type and ("id" in phpids_type or "lfi" in phpids_type)) and "../../" in urllib.unquote(attack_string)`
Filter or sanitization method used: `str_replace( array( "../"), "", $string)`
The solution is a "quick fix" for handling path traversal attacks and command injections with path traversals. The PHP function `str_replace()` replaces all occurrences of the search string ("../") with the replacement string ("")[3].

Fourth scenario: Attack not identified or not in the first three scenarios

`if`-condition to trigger scenario: `else` triggers the scenario.
Filter or sanitization method used: `preg_replace('/[â-zA-Z0-9]/', ", $string)`
When the attack is not identified or does not fall in either of the previous three categories, a uniform filtering is applied. `preg_replace()`[4] used in this context removes all characters which are not letters, numbers or dots (".").

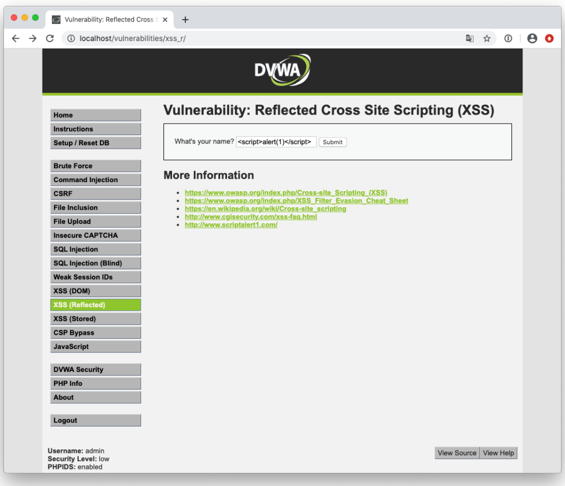
For the experiments, it is desirable to test each sanitization method. Therefore, experiments using specific exploits to trigger each scenario are conducted. The last experiment of step 4 will initiate scenario four for each exploit launched in the previous experiments. This is to compare how suitable distinct sanitization methods for self-healing are for our test environment in comparison to one being significantly more comprehensive.

Experiment 1 - XSS

In an XSS attack, malicious scripts are uploaded and run in the browser. In this exploit, the XSS is reflected and the input parameters **<script>alert(1)</script>**

[1]https://www.php.net/manual/en/function.htmlspecialchars.php
[2]https://www.php.net/manual/en/mysqli.real-escape-string.php
[3]https://www.php.net/manual/en/function.str-replace.php
[4]https://www.php.net/manual/en/function.preg-replace.php

are displayed in figure 5.53. The PHPIDS suggests that the attack is possibly one of the following types: **"xss csrf sqli rfe lfi"**.



Vulnerability: Reflected Cross Site Scripting (XSS)

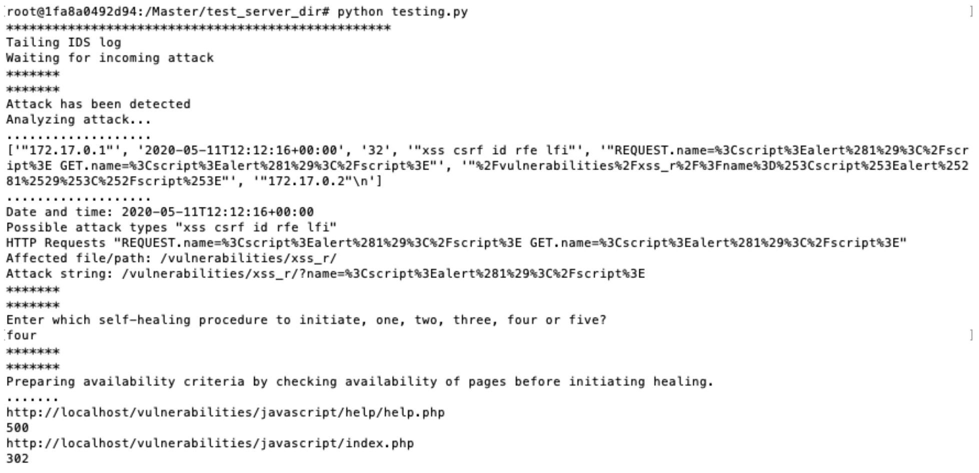
What's your name? <script>alert(1)</script>

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Username: admin
Security Level: low
PHPIDS: enabled

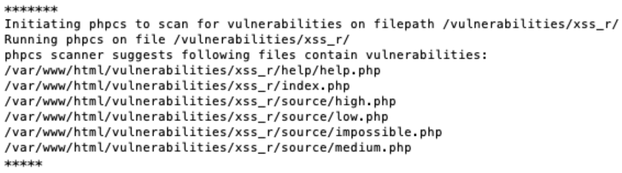
Figure 5.53: Example input of reflected XSS in experiment 1 step 4.



```
root@1fa8a0492d94:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-11T12:12:16+00:00', '32', '"xss csrf id rfe lfi"', '"REQUEST.name=%3Cscript%3Ealert%281%29%3C%2Fscr
ipt%3E GET.name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E"', '"%2Fvulnerabilities%2Fxss_r%2F%3Fname%3D%253Cscript%253Ealert%252
81%2529%253C%252Fscript%253E"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-11T12:12:16+00:00
Possible attack types "xss csrf id rfe lfi"
HTTP Requests "REQUEST.name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E GET.name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E"
Affected file/path: /vulnerabilities/xss_r/
Attack string: /vulnerabilities/xss_r/?name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
four
*******
*******
Preparing availability criteria by checking availability of pages before initiating healing.
.......
http://localhost/vulnerabilities/javascript/help/help.php
500
http://localhost/vulnerabilities/javascript/index.php
302
```

Figure 5.54: Reflected XSS detected and self-healing initiated in experiment 1 step 4.

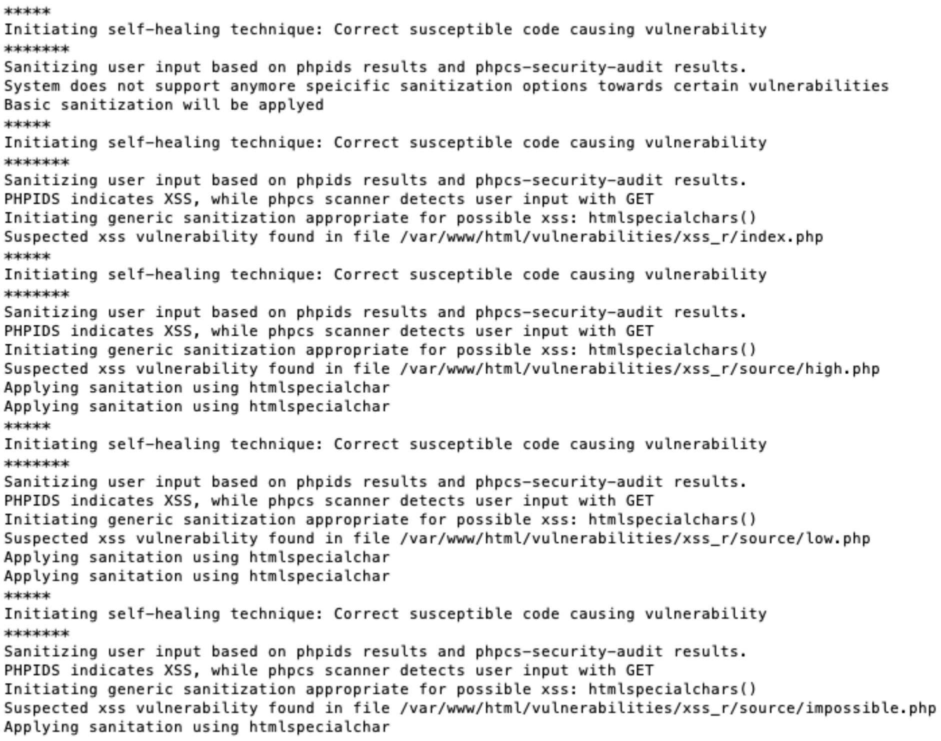
As the attack is detected, self-healing step 4 is initiated (figure 5.54) and phpcs-security-audit outputs its results for the vulnerable page (figure 5.55).



```
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/xss_r/
Running phpcs on file /vulnerabilities/xss_r/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/xss_r/help/help.php
/var/www/html/vulnerabilities/xss_r/index.php
/var/www/html/vulnerabilities/xss_r/source/high.php
/var/www/html/vulnerabilities/xss_r/source/low.php
/var/www/html/vulnerabilities/xss_r/source/impossible.php
/var/www/html/vulnerabilities/xss_r/source/medium.php
*****
```

Figure 5.55: phpcs-security-audit results for reflected XSS exploit in experiment 1 step 4.

Using the results from the scanner and the IDS, the self-healing can start. Figure 5.56 shows the output from healing the vulnerable files. The figure shows that the outputted, vulnerable files from the vulnerability scanner and the parameters from the IDS are used when choosing which of the four scenarios to use when healing.



```
*****
Initiating self-healing technique: Correct susceptible code causing vulnerability
*******
Sanitizing user input based on phpids results and phpcs-security-audit results.
System does not support anymore speicific sanitization options towards certain vulnerabilities
Basic sanitization will be applyed
*****
Initiating self-healing technique: Correct susceptible code causing vulnerability
*******
Sanitizing user input based on phpids results and phpcs-security-audit results.
PHPIDS indicates XSS, while phpcs scanner detects user input with GET
Initiating generic sanitization appropriate for possible xss: htmlspecialchars()
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/index.php
*****
Initiating self-healing technique: Correct susceptible code causing vulnerability
*******
Sanitizing user input based on phpids results and phpcs-security-audit results.
PHPIDS indicates XSS, while phpcs scanner detects user input with GET
Initiating generic sanitization appropriate for possible xss: htmlspecialchars()
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/source/high.php
Applying sanitation using htmlspecialchar
Applying sanitation using htmlspecialchar
*****
Initiating self-healing technique: Correct susceptible code causing vulnerability
*******
Sanitizing user input based on phpids results and phpcs-security-audit results.
PHPIDS indicates XSS, while phpcs scanner detects user input with GET
Initiating generic sanitization appropriate for possible xss: htmlspecialchars()
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/source/low.php
Applying sanitation using htmlspecialchar
Applying sanitation using htmlspecialchar
*****
Initiating self-healing technique: Correct susceptible code causing vulnerability
*******
Sanitizing user input based on phpids results and phpcs-security-audit results.
PHPIDS indicates XSS, while phpcs scanner detects user input with GET
Initiating generic sanitization appropriate for possible xss: htmlspecialchars()
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/source/impossible.php
Applying sanitation using htmlspecialchar
```

Figure 5.56: Self-healing procedure of reflected XSS in experiment 1 step 4.

Further examining one of the healed PHP files, `low.php`, listing 5.8 and 5.9

respectfully display the code before and after patching. The user input will now be sanitized using function `htmlspecialchars()`. The function is applied in line 6 and 7 of the file.

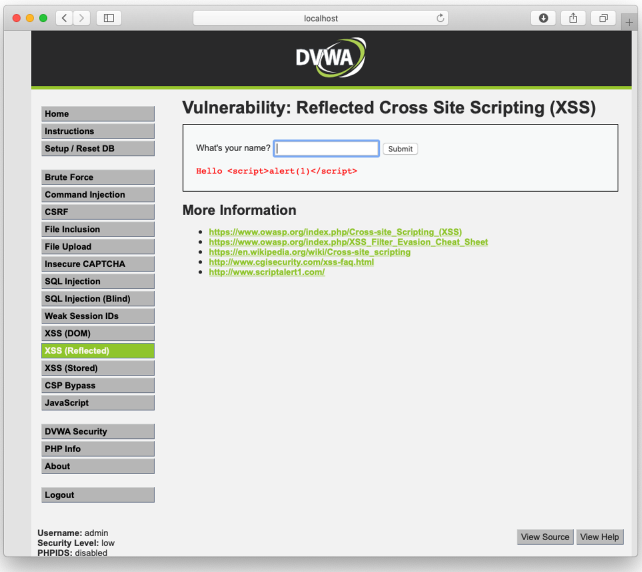
```
 1 <?php
 2 
 3 header ("X-XSS-Protection: 0");
 4 
 5 // Is there any input?
 6 if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
 7   // Feedback for end user
 8   $html .= '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
 9 }
10 
11 ?>
```

Listing 5.8: `low.php` before healing in experiment 1 step 4.

```
 1 <?php
 2 
 3 header ("X-XSS-Protection: 0");
 4 
 5 // Is there any input?
 6 if( array_key_exists( "name", $_GET ) && htmlspecialchars($_GET[ 'name'
      ]) != NULL ) {
 7   // Feedback for end user
 8   $html .= '<pre>Hello ' . htmlspecialchars($_GET[ 'name' ]) . '</pre>'
     ;
 9 }
10 
11 ?>
```

Listing 5.9: `low.php` after healing in experiment 1 step 4.

In order to evaluate whether the vulnerability is healed, manual inspection is needed. Relaunching of the exploit is illustrated in figure 5.57, and shows that the healing was a success since the browser did not execute `alert()` from the injected script. The special characters are converted to HTML entities by the sanitization function.



Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

```
Hello <script>alert(1)</script>
```

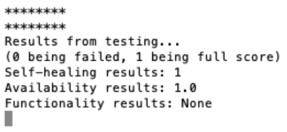
More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Username: admin
Security Level: low
PHPIDS: disabled

Figure 5.57: Browser result from relaunching exploit after healing in experiment 1 step 4.

The results from the availability test, self-healing test and functionality test from the script are presented in figure 5.58. Since the script has not automated functionality testing for the page with the reflected XSS vulnerability (`/vulnerabilities/xss_r/`), the results for this test shows "`None`". However, we can conclude that the functionality is preserved after healing by manual inspection; the functionality of the page is simply to display to the user what is inputted in the form, illustrated in figure 5.57. From the results, we can also read that the availability was not distorted from healing.

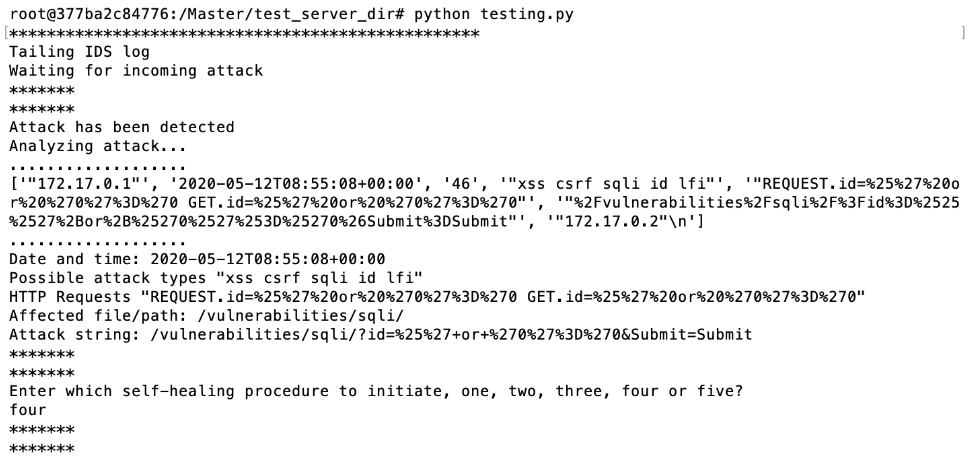


```
********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: None
█
```

Figure 5.58: Results from experiment 1 step 4.

Experiment 2 - SQL injection

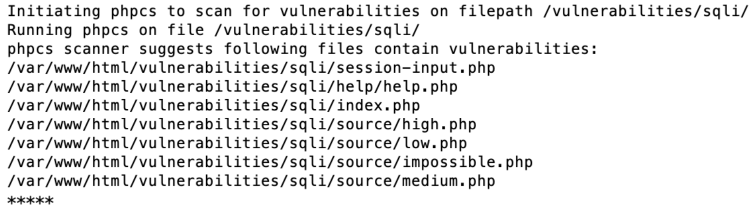
Using the input as in previous experiments, **"%' or '0'='0"** to trigger the IDS, the self-healing responds as presented in figure 5.59. The attack is, according to the IDS, identified as one of the following: **"xss csrf sqli id lfi"**.



```
root@377ba2c84776:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-12T08:55:08+00:00', '46', '"xss csrf sqli id lfi"', '"REQUEST.id=%25%27%20o
r%20%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"', '"%2Fvulnerabilities%2Fsqli%2F%3Fid%3D%2525
%2527%2Bor%2B%25270%2527%253D%25270%26Submit%3DSubmit"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-12T08:55:08+00:00
Possible attack types "xss csrf sqli id lfi"
HTTP Requests "REQUEST.id=%25%27%20or%20%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"
Affected file/path: /vulnerabilities/sqli/
Attack string: /vulnerabilities/sqli/?id=%25%27+or+%270%27%3D%270&Submit=Submit
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
four
*******
*******
```

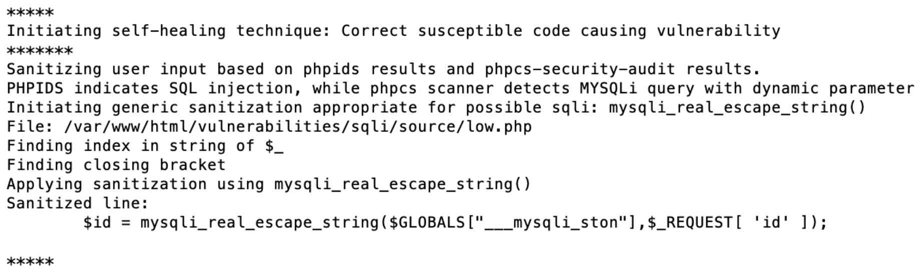
Figure 5.59: SQL injection detected by self-healing script in experiment 2 step 4.

The results from the vulnerability scanner are equal to previous scans of `/vulnerabilities/sqli/` (figure 5.60), and are used in the self-healing technique. The script identifies the attack as an SQL injection, and uses `mysqli_real_escape_string()` to sanitize the user input. Figure 5.61 illustrates an example from the healing procedure, with `low.php` as example file.



```
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/sqli/
Running phpcs on file /vulnerabilities/sqli/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/sqli/session-input.php
/var/www/html/vulnerabilities/sqli/help/help.php
/var/www/html/vulnerabilities/sqli/index.php
/var/www/html/vulnerabilities/sqli/source/high.php
/var/www/html/vulnerabilities/sqli/source/low.php
/var/www/html/vulnerabilities/sqli/source/impossible.php
/var/www/html/vulnerabilities/sqli/source/medium.php
*****
```

Figure 5.60: phpcs-security-audit results in experiment 2 step 4.



```
*****
Initiating self-healing technique: Correct susceptible code causing vulnerability
*******
Sanitizing user input based on phpids results and phpcs-security-audit results.
PHPIDS indicates SQL injection, while phpcs scanner detects MYSQLi query with dynamic parameter
Initiating generic sanitization appropriate for possible sqli: mysqli_real_escape_string()
File: /var/www/html/vulnerabilities/sqli/source/low.php
Finding index in string of $_
Finding closing bracket
Applying sanitization using mysqli_real_escape_string()
Sanitized line:
        $id = mysqli_real_escape_string($GLOBALS["___mysqli_ston"],$_REQUEST[ 'id' ]);

*****
```

Figure 5.61: Self-healing `low.php` in experiment 2 step 4.

Further inspection of the source code of `low.php` before and after healing in listing 5.10 and 5.11, the changes are featured. The fifth line in the code sanitizes the user input before it is used in the SQL query.

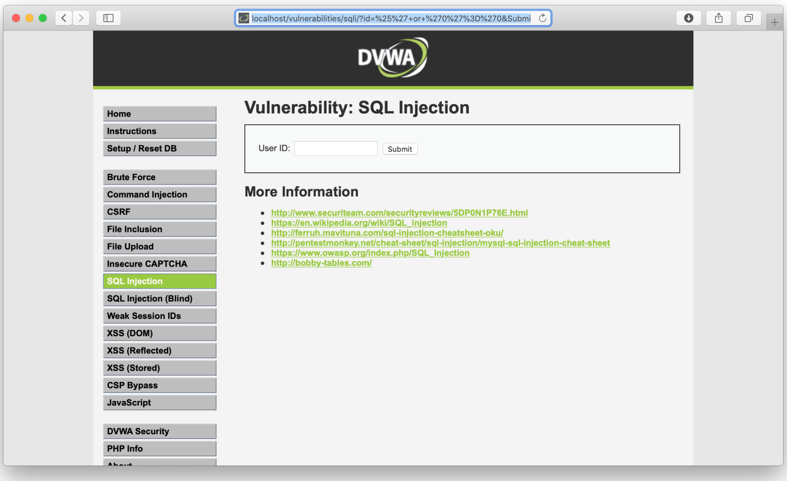
```
1  <?php
2
3  if( isset( $_REQUEST[ 'Submit' ] ) ) {
4    // Get input
5    $id = $_REQUEST[ 'id' ];
6
7    // Check database
8    $query  = "SELECT first_name, last_name FROM users WHERE user_id = '
       $id';";
9    $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die(
       '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error(
       $GLOBALS["___mysqli_ston"]) : (($___mysqli_res =
       mysqli_connect_error()) ? $___mysqli_res : false)) . '</pre>' );
10
11   // Get results
12   while( $row = mysqli_fetch_assoc( $result ) ) {
13     // Get values
14     $first = $row["first_name"];
15     $last  = $row["last_name"];
16
17     // Feedback for end user
18     $html .= "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {
       $last}</pre>";
19   }
20
21   mysqli_close($GLOBALS["___mysqli_ston"]);
22 }
23
24 ?>
```

Listing 5.10: `low.php` before healing in experiment 2 step 4.

```php
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
  // Get input
  $id = mysqli_real_escape_string($GLOBALS["___mysqli_ston"],$_REQUEST[
     'id' ]);

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

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

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

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

?>
```

Listing 5.11: `low.php` after healing in experiment 2 step 4.

To evaluate if the self-healing was a success, the attack is rerun. The browser display when doing so, is shown in figure 5.62. The malicious attack string is sanitized and therefore does not interrupt the SQL query.



localhost/vulnerabilities/sqli/?id=%25%27+or+%270%27%3D%270&Submi

Vulnerability: SQL Injection

User ID:

More Information

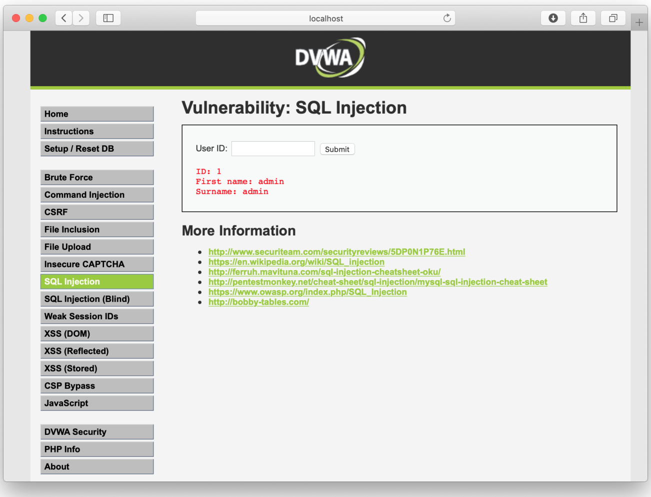
- http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- https://en.wikipedia.org/wiki/SQL_injection
- http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
- https://www.owasp.org/index.php/SQL_Injection
- http://bobby-tables.com/

Figure 5.62: Example in browser after healing in experiment 2 step 4.

For the functionality test, the results are displayed in figure 5.63 and 5.64. These show that the functionality of the page is kept in accordance to the unit test; the page outputs the ID which is requested, namely **"admin"**.

```
*******
Initiating functionality test for file /vulnerabilities/sqli/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/sqli/, the expected behavior is to display ID, first name and surname when given
 an ID as input.
<pre>ID: 1<br />First name: admin<br />Surname: admin</pre> found in output.
The functionality of /sqli/ has been preserved after healing.
```

Figure 5.63: Functionality results in experiment 2 step 4.



Vulnerability: SQL Injection

```
ID: 1
First name: admin
Surname: admin
```

More Information

- http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- https://en.wikipedia.org/wiki/SQL_injection
- http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
- https://www.owasp.org/index.php/SQL_Injection
- http://bobby-tables.com/

Figure 5.64: Example in browser for functionality test after healing in experiment 2 step 4.

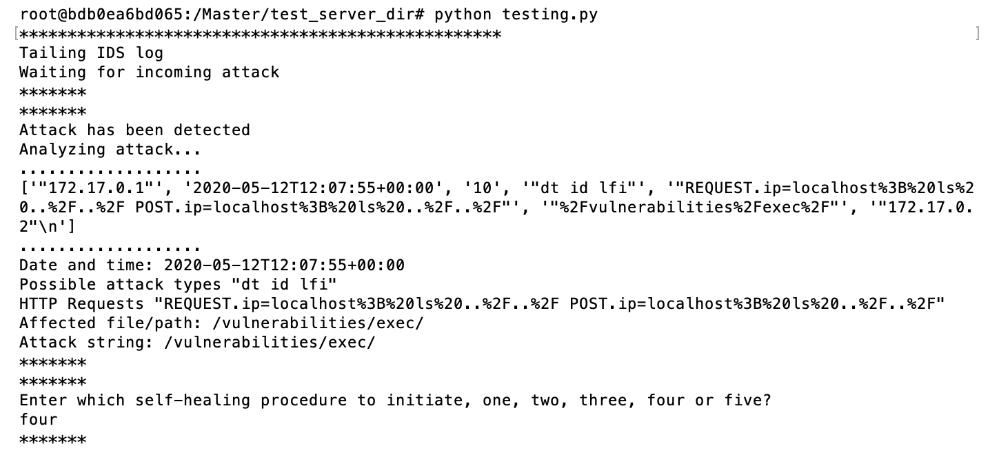
The final results from the experiment are summarized in figure 5.65 from the self-healing script. The experiment achieves a complete score on each evaluation criteria.

```
********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: 1
```

Figure 5.65: Results in experiment 2 step 4.

Experiment 3 - Command injection

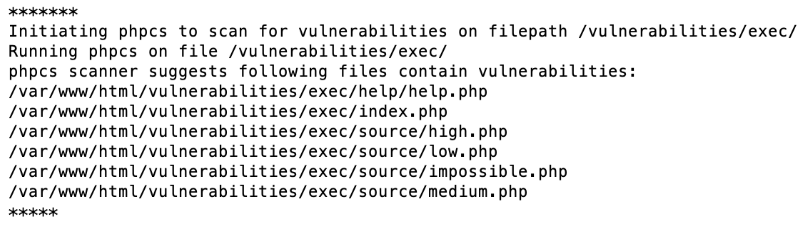
The experiment used **"localhost; ls ../../"** to trigger the self-healing script, illustrated in figure 5.66. The IDS indicates that the attack is one of the following **"dt id lfi"**.



```
root@bdb0ea6bd065:/Master/test_server_dir# python testing.py
*************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
..................
['"172.17.0.1"', '2020-05-12T12:07:55+00:00', '10', '"dt id lfi"', '"REQUEST.ip=localhost%3B%20ls%2
0..%2F..%2F POST.ip=localhost%3B%20ls%20..%2F..%2F"', '"%2Fvulnerabilities%2Fexec%2F"', '"172.17.0.
2"\n']
..................
Date and time: 2020-05-12T12:07:55+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.ip=localhost%3B%20ls%20..%2F..%2F POST.ip=localhost%3B%20ls%20..%2F..%2F"
Affected file/path: /vulnerabilities/exec/
Attack string: /vulnerabilities/exec/
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
four
*******
```

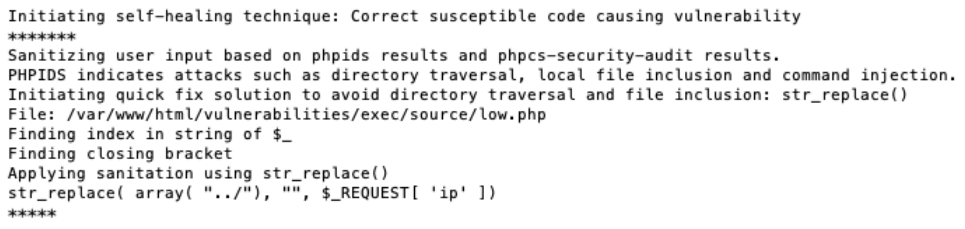
Figure 5.66: Self-healing script initiated by command injection attempt in experiment 3 step 4.

The vulnerability scanner outputs which files are vulnerable (figure 5.67), and these are used in the self-healing. Healing of one of the vulnerable files, `low.php`, is presented in figure 5.68. We can see from the figure that the self-healing procedure initiates healing for a directory traversal and/or command injection vulnerability.



```
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/exec/
Running phpcs on file /vulnerabilities/exec/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/exec/help/help.php
/var/www/html/vulnerabilities/exec/index.php
/var/www/html/vulnerabilities/exec/source/high.php
/var/www/html/vulnerabilities/exec/source/low.php
/var/www/html/vulnerabilities/exec/source/impossible.php
/var/www/html/vulnerabilities/exec/source/medium.php
*****
```

Figure 5.67: phpcs-security-audit results in experiment 3 step 4.



```
Initiating self-healing technique: Correct susceptible code causing vulnerability
*******
Sanitizing user input based on phpids results and phpcs-security-audit results.
PHPIDS indicates attacks such as directory traversal, local file inclusion and command injection.
Initiating quick fix solution to avoid directory traversal and file inclusion: str_replace()
File: /var/www/html/vulnerabilities/exec/source/low.php
Finding index in string of $_
Finding closing bracket
Applying sanitation using str_replace()
str_replace( array( "../"), "", $_REQUEST[ 'ip' ])
*****
```

Figure 5.68: Example from healing procedure in step 3 step 4.

The source code for `low.php` before and after healing is shown in listing 5.12 and 5.13. The fifth line of code shows where modifications have been made, hence where `str_replace` has been implemented after healing.

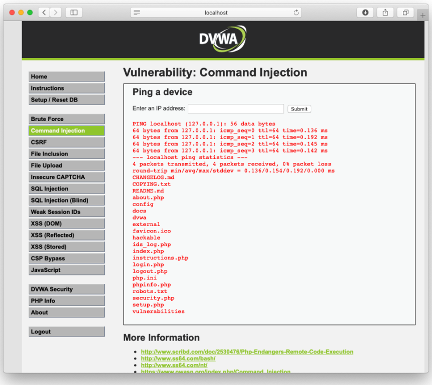
```
1  <?php
2  
3  if( isset( $_POST[ 'Submit' ]  ) ) {
4      // Get input
5      $target = $_REQUEST[ 'ip' ];
6  
7      // Determine OS and execute the ping command.
8      if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
9          // Windows
10         $cmd = shell_exec( 'ping  ' . $target );
11     }
12     else {
13         // *nix
14         $cmd = shell_exec( 'ping  -c 4 ' . $target );
15     }
16 
17     // Feedback for the end user
18     echo "<pre>{$cmd}</pre>";
19 }
```

Listing 5.12: `low.php` before healing in experiment 3 step 4.

```
1  <?php
2  
3  if( isset( $_POST[ 'Submit' ]  ) ) {
4      // Get input
5      $target = str_replace( array( "../"), "", $_REQUEST[ 'ip' ]);
6  
7      // Determine OS and execute the ping command.
8      if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}
```

Listing 5.13: `low.php` after healing in experiment 3 step 4.

When evaluating if the vulnerability has been removed, the original exploit was relaunched. The results in a browser before and after healing are shown in figure 5.69 and 5.70. The healing removed the **"../../"** used in the attack, but it appears that the command injection vulnerability is still present since the command **ls** is executed. This can cause concern for other attacks not relying on traversing directories. Figure 5.71 shows another, classic command injection string, namely **"cat /etc/passwd"**, being executed successfully on the server after healing. This strongly indicates that the healing was not a success.



Vulnerability: Command Injection

Ping a device

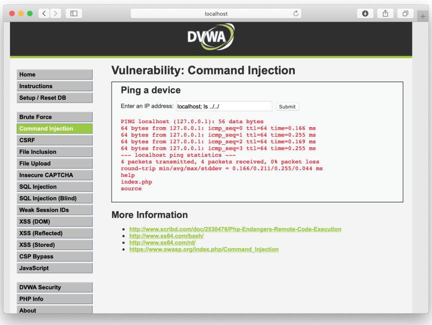
Enter an IP address:

```
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.136 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.192 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.145 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.142 ms
--- localhost ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.136/0.154/0.192/0.000 ms
CHANGELOG.md
COPYING.txt
README.md
about.php
config
docs
dvwa
external
favicon.ico
hackable
ids_log.php
index.php
instructions.php
login.php
logout.php
php.ini
phpinfo.php
robots.txt
security.php
setup.php
vulnerabilities
```

More Information

- http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
- http://www.ss64.com/bash/
- http://www.ss64.com/nt/
- https://www.owasp.org/index.php/Command_Injection

Figure 5.69: Launched command injection attack before healing in step 3 step 4.



Vulnerability: Command Injection

Ping a device

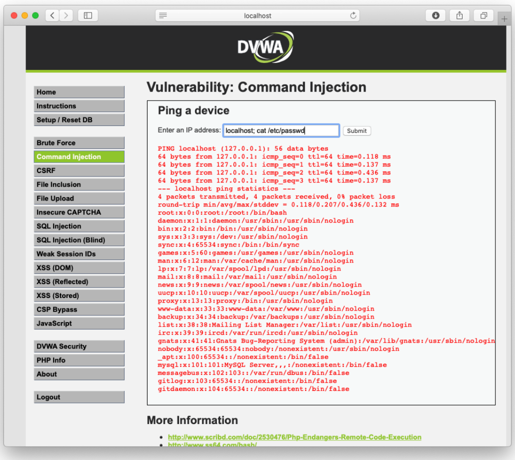
Enter an IP address: localhost; ls ../../

```
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.166 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.255 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.169 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.255 ms
--- localhost ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.166/0.211/0.255/0.044 ms
help
index.php
source
```

More Information

- http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
- http://www.ss64.com/bash/
- http://www.ss64.com/nt/
- https://www.owasp.org/index.php/Command_Injection

Figure 5.70: Launched command injection attack after healing in step 3 step 4.



Vulnerability: Command Injection

Ping a device

Enter an IP address: localhost; cat /etc/passwd

```
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.118 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.137 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.436 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.137 ms
--- localhost ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.118/0.207/0.436/0.132 ms
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false
mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false
messagebus:x:102:103::/var/run/dbus:/bin/false
gitlog:x:103:65534::/nonexistent:/bin/false
gitdaemon:x:104:65534::/nonexistent:/bin/false
```

More Information

- http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
- http://www.ss64.com/bash/

Figure 5.71: Second command injection attack after healing in step 3 step 4.

Therefore, the self-healing criteria is not met. For the functionality testing, the page is still able to ping the given URL, resulting in full score. The two tests are illustrated in figure 5.72. The summary of the results are given in figure 5.73.

```
*******
Initiating self-healing test for self-healing technique ...
*********
Relaunch exploit manually with given attack url and parameter to validate if vulnerability is unrea
chable/healed:
URL: http://localhost/vulnerabilities/exec/
Input: localhost; ls ../../
*********
Press y or yes to confirm healing
n

*******
Initiating functionality test for file /vulnerabilities/exec/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/exec, the expected behavior is to ping given ip or url.
```

Figure 5.72: Functionality test results in step 3 step 4.

```
</html>
The functionality of the page exec is preserved

********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 0
Availability results: 1.0
Functionality results: 1
```

Figure 5.73: Results from tests in step 3 step 4.

Experiment 4 - File inclusion

The experiment possesses portions of the same results as the previous experiment. The attack string used is **"../../../../etc/passwd"**, which results in PHPIDS classifying it as either **"dt id lfi"**. The self-healing procedure recognizes the attack pattern as either command injection, directory traversal or file inclusion. Using the file `low.php` as an example from the healing, its source code before and after healing is presented in respectfully listing 5.14 and 5.15. The `$file` variable is used in the `index.php`, presented in listing 5.16.

```php
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

?>
```

Listing 5.14: `low.php` before healing in experiment 4 step 4.

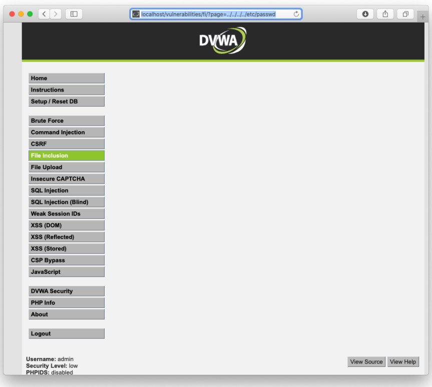
```php
<?php

// The page we wish to display
$file = str_replace( array( "../"), "", $_GET[ 'page' ]);

?>
```

Listing 5.15: `low.php` after healing in experiment 4 step 4.

```php
...
if( isset( $file ) )
  include( $file );
else {
  header( 'Location:?page=include.php' );
  exit;
}
...
```

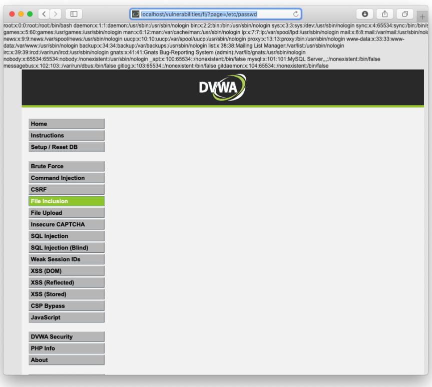
Listing 5.16: Parts of `index.php` which includes files on the server in experiment 4 step 4.

The browser display when launching an exploit using the original attack string and a modification (**"/etc/passwd/"**) after healing are presented in figure 5.74 and 5.75. When inspecting these, the same results as in the command injection experiment are prevalent. It is still possible to include files even though directory traversal with **"../../"** is infeasible.



Username: admin
Security Level: low
PHPIDS: disabled

Figure 5.74: File inclusion attempt after healing in experiment 4 step 4.



```
root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/s
games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nolo
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-
data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin _apt:x:100:65534::/nonexistent:/bin/false mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false
messagebus:x:102:103::/var/run/dbus:/bin/false gitlog:x:103:65534::/nonexistent:/bin/false gitdaemon:x:104:65534::/nonexistent:/bin/false
```

Figure 5.75: Second file inclusion attack after healing in experiment 4 step 4.

Therefore, healing of the vulnerability was not a success. However, both availability and functionality are preserved after healing. The results are, therefore, equivalent to the previous experiment, summarized in figure 5.73.

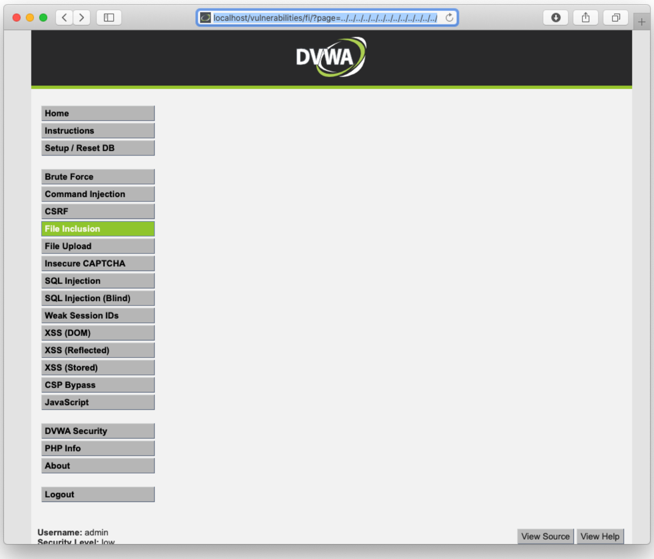
Experiment 5 - Extensive sanitization

The last scenario of healing technique step 4 is a general, extensive mechanism that removes all characters except for letters, numbers, and dots. To trigger this process, the technique must not associate the attack parameters with either one of the previous three scenarios. However, in this experiment, the healing mechanism is used on the former four attack schemes to evaluate its performance. In addition, a stored XSS is tested in this experiment. A summary of the details of the conducted experiment is shown in table 5.4.

Path	**Vulnerability/ exploit**	**Attack input**
/vulnerabilities/xss_r/	Reflected XSS	<script>alert(1)</script>
/vulnerabilities/sqli/	SQL injection	%' or '0'='0
/vulnerabilities/exec/	Command injection	localhost; ls ../../
/vulnerabilities/fi/	Local file inclusion	../../../../../../../../../../ ../../etc/passwd
/vulnerabilities/xss_s/	Stored XSS	<script>alert(1)</script>

Table 5.4: Table summarizing attacks to trigger scenario four of step four.

After healing the five exploits using `preg_replace()` to replace all non-letters, non-integers and non-dots, the vulnerabilities were not any more reachable and therefore not exploitable for the given attack inputs. The results from the file inclusion is presented in figure 5.76 and 5.77. The images show that the illegal characters are removed from the input before executing the request. The patched source code of `/vulnerabilities/fi/source/low.php` in listing 5.17 illustrates how `preg_replace()` has been implemented.


localhost/vulnerabilities/fi/?page=../../../../../../../../../../../

Username: admin
Security Level: low

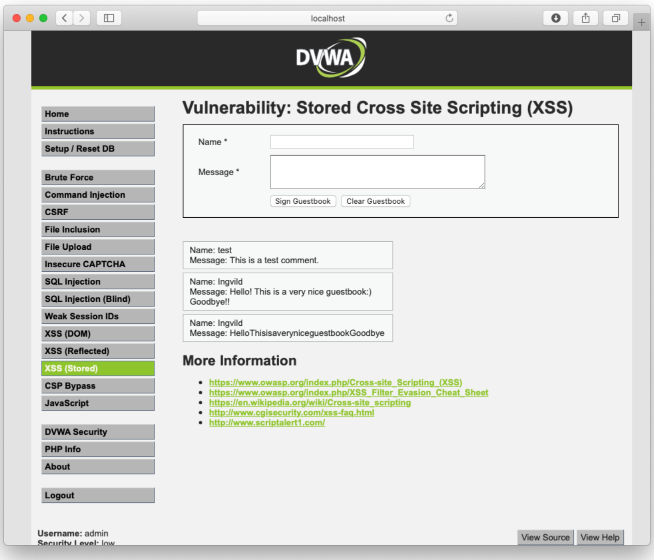
Figure 5.76: Launching file inclusion attack in experiment 5 step 4.

```
[Thu May 14 05:30:28.894869 2020] [:error] [pid 338] [client 172.17.0.1:54470]
PHP Warning:  include(): Failed opening '..........................etcpasswd' f
or inclusion (include_path='.:/usr/share/php:../../external/phpids/0.6/lib/') i
n /var/www/html/vulnerabilities/fi/index.php on line 36
```

Figure 5.77: Error in apache2 error log from file inclusion attempt in experiment 5 step 4.

```php
<?php

// The page we wish to display
$file = preg_replace('/[^a-zA-Z0-9.]/', '', $_GET[ 'page' ]);

?>
```

Listing 5.17: `/vulnerabilities/fi/source/low.php` after healing in experiment 5 step 4.



Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

Name: test
Message: This is a test comment.

Name: Ingvild
Message: Hello! This is a very nice guestbook:)
Goodbye!!

Name: Ingvild
Message: HelloThisisaveryniceguestbookGoodbye

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Username: admin
Security Level: low

Figure 5.78: Stored XSS functionality deviation in experiment 5 step 4.

The availability of all pages was also kept. Functionality was preserved, but there are cases showing that the functionality is not entirely as it was before healing. Figure 5.78 shows an example from the stored XSS with a comment section with

	Self-healing test	**Availability test**	**Functionality test**	**Average**
Exp. 1	1	1	None	1
Exp. 2	1	1	1	1
Exp. 3	0	1	1	0.666
Exp. 4	0	1	1	0.666
Exp. 5	1	1	1	1
Average	0.6	1	1	0.867

Table 5.5: Summary of results for step 4. The first five rows give the results from each experiments, including the average performance of each experiment in the final column. The fifth experiment includes the result from the average of each subexperiments. The last row gives the average results from all experiments, and the bottom right cell gives an idea of how well the self-healing technique performed concerning the evaluation criteria. The table shows that there was some differences in performance. However, according to the given evaluation criteria, the self-healing technique achieved the best scores for experiment 1, 2 and 5.

the same comment before and after healing. Even though the vulnerability has been removed after healing, it shows that the guestbook displays comments differently because of the sanitization.

Summary of results

In this self-healing technique, popular quick fixes and sanitization methods for a variety of PHP web vulnerabilities have been tested. When relaunching the attacks which triggered the IDS in experiment 1, 2, 3, and 4, the self-healing criteria was met for all but experiment 3. However, variations of attack input in experiment 4 proved that the sanitization method used is not sufficient with regards to healing. For experiment 2, the use of `mysqli_real_escape_string()` to prevent SQL injections is discredited, since it is often misapplied. In this self-healing script, scenario 2 using this sanitization method does not make sure that the variable is quoted correctly in the SQL query. If this was not priorly implemented in the PHP file, the healing mechanism leaves the page still vulnerable to SQL injections.

The fifth experiment comprises tests of several vulnerabilities after being healed with a comprehensive sanitization method. The results show that the healing was successful for all the vulnerabilities, but at the expense of server functionality. The experimental results are presented in table 5.5.

5.5 Step 5 - Correct susceptible code causing vulnerability

In the last self-healing procedure, the goal is to correct the code in which inflicts vulnerabilities. This stands out from the other, former steps since it tries to remove concrete vulnerabilities using recommended patching, rather than eliminating symptoms of breaches. It currently attempts to identify XSSs, command injection using `shell_exec()` and SQL injections. The method does not use wide-ranging patching suitable for multiple vulnerabilities such as the generic approach of step 4. It has a specific patching alternative for the given vulnerability, narrowed down to specific susceptible functions and appliances.

XSS patching

`if`-condition to trigger patching: `("User input" and postget) in '\t'.join(phpcs_error_messages) or "HTML construction with direct user input" in '\t'.join(phpcs_error_messages) and "xss" in phpids_type`
Patching approach: `htmlspecialchars()`
Follows the same reasoning as the first scenario in step 4.

Command injection with `shell_exec()` patching

`if`-condition to trigger patching: `"shell_exec" in '\t'.join(phpcs_error_messages) and ("dt" in phpids_type or "id" in phpids_type or "lfi" in phpids_type)`
Patching approach: `shell_exec('cmd ' . escapeshellarg($string))`
The PHP function `escapeshellarg()` escapes a string to be used as a shell argument[5], preventing disruptions of shell command execution.

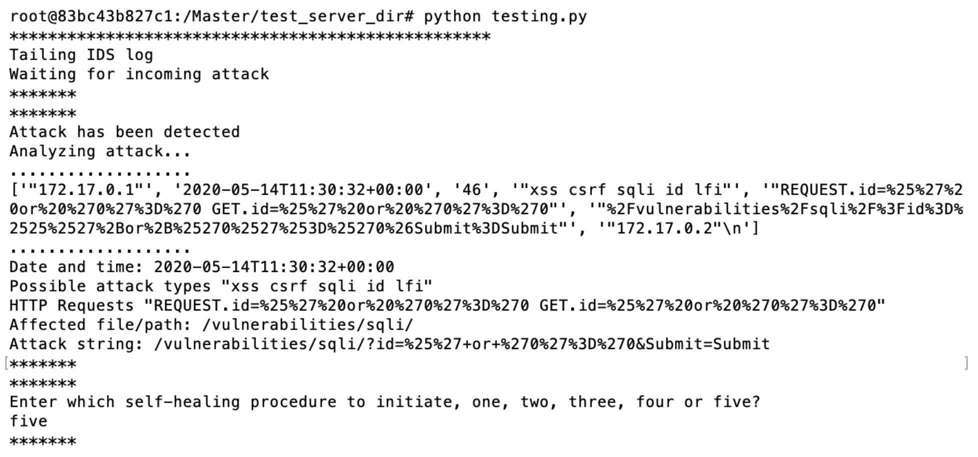
SQL injection patching

`if`-condition to trigger patching: `"sqli" in phpids_type and "MYSQLi" in '\t'.join(phpcs_error_messages)`
Patching approach: Prepared statements
One of the most efficient solutions to avoiding SQL injections in PHP are by the use of prepared statements[6,7]. This is implemented in the SQL injection-specific patching of step 5. The protection is achieved by using `mysqli_prepare()` to prepare the statement for further processing, using it in the function `mysqli_stmt_bind_param()` to bind the parameters and executing the query using `mysqli_stmt_execute()`. `mysqli_stmt_get_result()` is used to receive the results from the query.

[5]https://www.php.net/manual/en/function.escapeshellarg.php
[6]https://www.php.net/manual/en/security.database.sql-injection.php
[7]https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

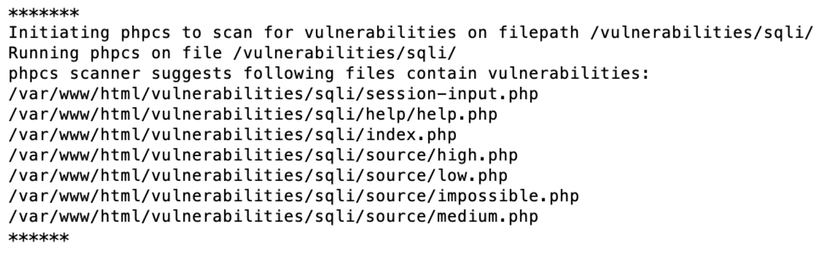
Experiment 1 - SQL injection

Using input **"%' or '0'='0"** to trigger the IDS and self-healing script, the display from the script is illustrated in figure 5.79. The IDS suggests that attack is either one of **"xss csrf sqli id lfi"**. The vulnerability scanner is run (figure 5.80), and the results are equivalent to earlier runs of the scanner.



```
root@83bc43b827c1:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-14T11:30:32+00:00', '46', '"xss csrf sqli id lfi"', '"REQUEST.id=%25%27%2
0or%20%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"', '"%2Fvulnerabilities%2Fsqli%2F%3Fid%3D%
2525%2527%2Bor%2B%25270%2527%253D%25270%26Submit%3DSubmit"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-14T11:30:32+00:00
Possible attack types "xss csrf sqli id lfi"
HTTP Requests "REQUEST.id=%25%27%20or%20%270%27%3D%270 GET.id=%25%27%20or%20%270%27%3D%270"
Affected file/path: /vulnerabilities/sqli/
Attack string: /vulnerabilities/sqli/?id=%25%27+or+%270%27%3D%270&Submit=Submit
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
five
*******
```

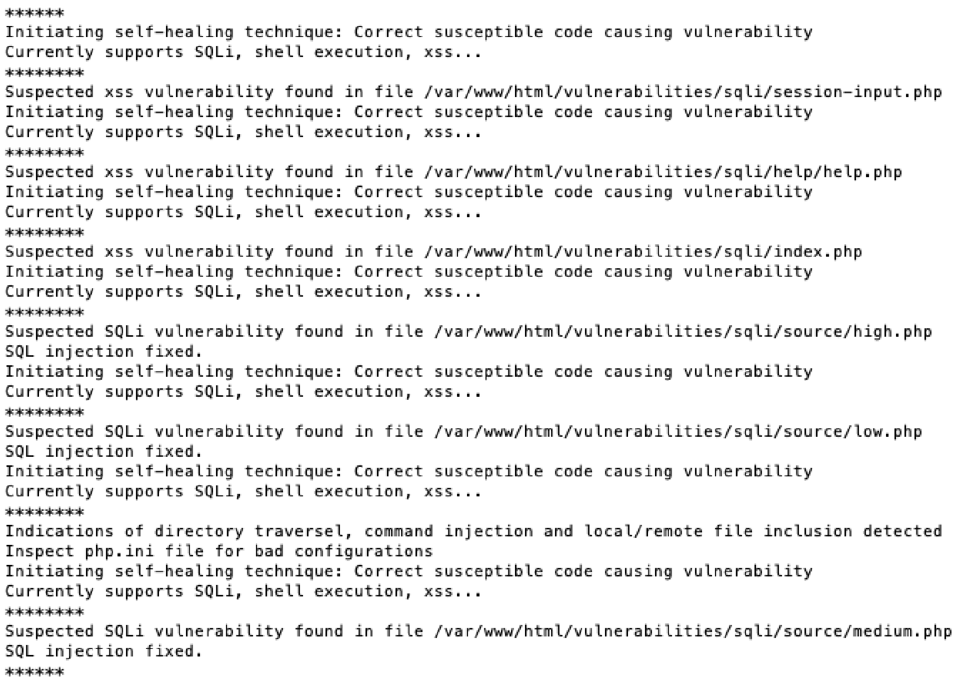
Figure 5.79: SQL injection detected and healing technique 5 is selected in experiment 1 step 5.



```
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/sqli/
Running phpcs on file /vulnerabilities/sqli/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/sqli/session-input.php
/var/www/html/vulnerabilities/sqli/help/help.php
/var/www/html/vulnerabilities/sqli/index.php
/var/www/html/vulnerabilities/sqli/source/high.php
/var/www/html/vulnerabilities/sqli/source/low.php
/var/www/html/vulnerabilities/sqli/source/impossible.php
/var/www/html/vulnerabilities/sqli/source/medium.php
******
```

Figure 5.80: phpcs-security-audit results in experiment 1 step 5.

Each of the vulnerable files are iterated through in the self-healing process to determine what kind of vulnerability might be present, as can be seen in figure 5.81.



```
******
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/sqli/session-input.php
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/sqli/help/help.php
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/sqli/index.php
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected SQLi vulnerability found in file /var/www/html/vulnerabilities/sqli/source/high.php
SQL injection fixed.
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected SQLi vulnerability found in file /var/www/html/vulnerabilities/sqli/source/low.php
SQL injection fixed.
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Indications of directory traversel, command injection and local/remote file inclusion detected
Inspect php.ini file for bad configurations
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected SQLi vulnerability found in file /var/www/html/vulnerabilities/sqli/source/medium.php
SQL injection fixed.
******
```

Figure 5.81: Healing of all vulnerable files in experiment 1 step 5.

The file which we launched the SQL injection towards, `low.php`, is suspected to contain such a vulnerability. The code of the file before and after healing is presented in listing 5.18 and 5.19. Line 8 and 9 of script 5.18 are modified to line 8 to 12 in 5.19.

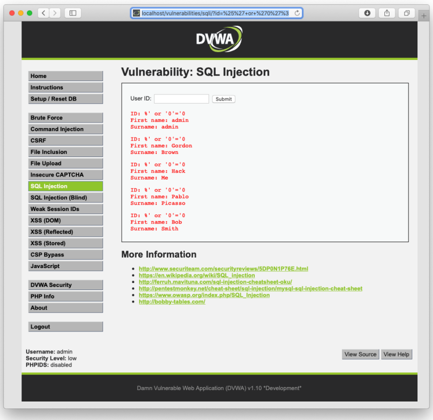
```
1  <?php
2
3  if( isset( $_REQUEST[ 'Submit' ] ) ) {
4    // Get input
5    $id = $_REQUEST[ 'id' ];
6
7    // Check database
8    $query  = "SELECT first_name, last_name FROM users WHERE user_id = '
       $id';";
9    $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die(
       '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error(
       $GLOBALS["___mysqli_ston"]) : (($___mysqli_res =
       mysqli_connect_error()) ? $___mysqli_res : false)) . '</pre>' );
10
11   // Get results
12   while( $row = mysqli_fetch_assoc( $result ) ) {
    // Get values
    $first = $row["first_name"];
    $last  = $row["last_name"];

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

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

?>
```

Listing 5.18: `low.php` before healing SQL injection in experiment 1 step 5.

```
<?php

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

    // Check database
    $query  = "SELECT first_name, last_name FROM users WHERE user_id =
    ?";
    $prep = mysqli_prepare($GLOBALS["___mysqli_ston"],    $query);
    mysqli_stmt_bind_param($prep,'s',    $id);
    mysqli_stmt_execute($prep);
    $result =mysqli_stmt_get_result($prep);

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

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

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

?>
```

Listing 5.19: `low.php` after healing SQL injection in experiment 1 step 5.

When testing if the SQL injection vulnerability is removed, the same attack that triggered the IDS is launched. Figure 5.82 shows the output in browser before healing, while figure 5.83 shows after healing. The SQL injection has been removed, as indicated in the figures.



localhost/vulnerabilities/sqli/?id=%25%27+or+%270%27%3

Vulnerability: SQL Injection

User ID:

```
ID: %' or '0'='0
First name: admin
Surname: admin

ID: %' or '0'='0
First name: Gordon
Surname: Brown

ID: %' or '0'='0
First name: Hack
Surname: Me

ID: %' or '0'='0
First name: Pablo
Surname: Picasso

ID: %' or '0'='0
First name: Bob
Surname: Smith
```

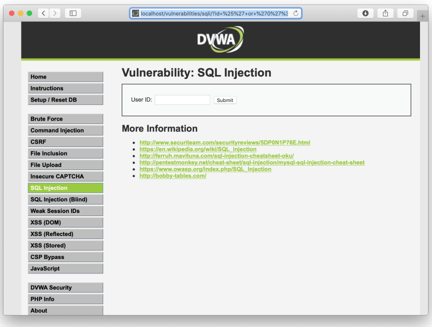
More Information

- http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- https://en.wikipedia.org/wiki/SQL_injection
- http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
- https://www.owasp.org/index.php/SQL_Injection
- http://bobby-tables.com/

Username: admin
Security Level: low
PHPIDS: disabled

Damn Vulnerable Web Application (DVWA) v1.10 *Development*

Figure 5.82: Browser output from SQL injection attack before healing in experiment 1 step 5.



localhost/vulnerabilities/sqli/?id=%25%27+or+%270%27%3

Vulnerability: SQL Injection

User ID:

More Information

- http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- https://en.wikipedia.org/wiki/SQL_injection
- http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
- https://www.owasp.org/index.php/SQL_Injection
- http://bobby-tables.com/

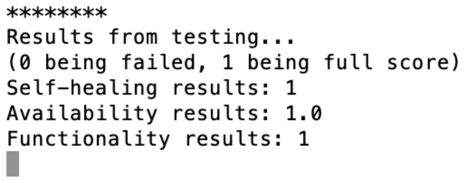
Figure 5.83: Browser output from SQL injection attack after healing in experiment 1 step 5.

After having healed the vulnerability, measurement of functionality criteria is initiated. Figure 5.84 presents the functionality test and its results from the self-healing script. The script was able to heal the vulnerability while maintaining essential functionality of the page.

```
*******
Initiating functionality test for file /vulnerabilities/sqli/
Currently supports unit/regression tests for /exec/, /fi/ and /sqli/
******
Running authenticating procedure to server...
... Authenticated
For /vulnerabilities/sqli/, the expected behavior is to display ID, first name and surname when giv
en an ID as input.
<pre>ID: 1<br />First name: admin<br />Surname: admin</pre> found in output.
The functionality of /sqli/ has been preserved after healing.

********
```

Figure 5.84: Functionality test results in experiment 1 step 5.

The final results from the three evaluation measurements are summarized in figure 5.85. Each criteria achieves the same result, being **1**.

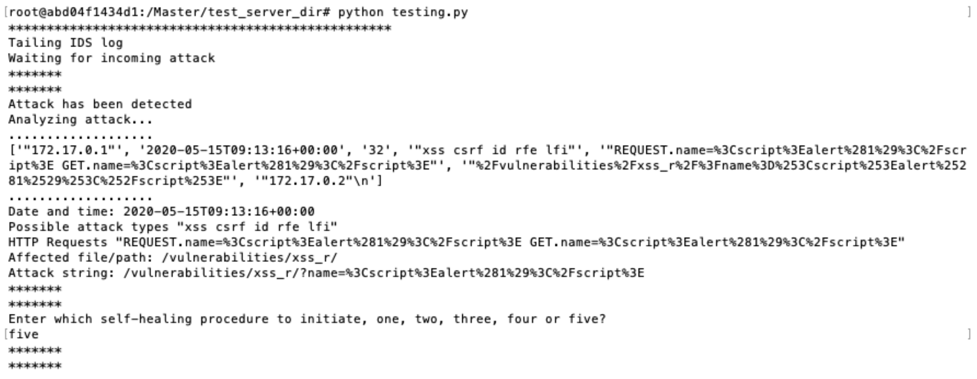


```
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: 1
```

Figure 5.85: Results from experiment 1 step 5.

Experiment 2 - XSS

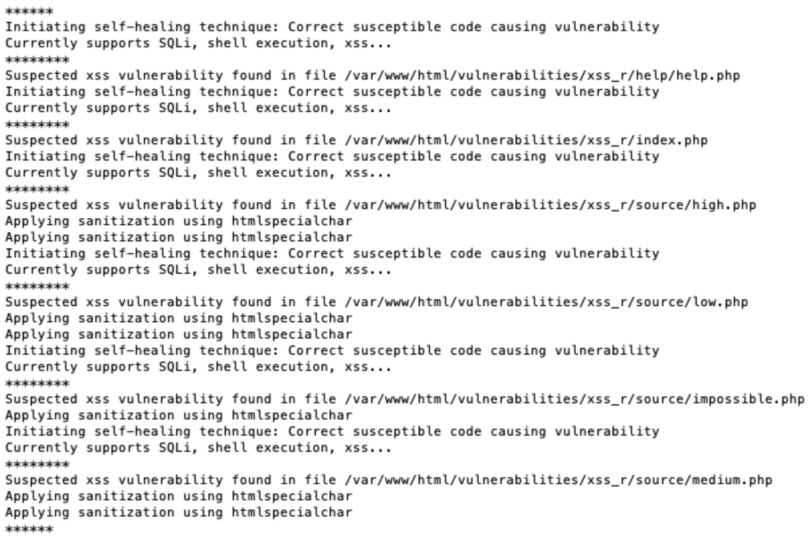
For the experiment, input **<script>alert(1)</script>** was used to initiate the self-healing script. The experiment was conducted on both pages with a reflected XSS vulnerability and stored XSS vulnerability, using the same input. The IDS yields the same predictions for possible attacks: **"xss csrf id rfe lfi"**. The reflected XSS details from when it was detected are displayed in figure 5.86.



```
root@abd04f1434d1:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-15T09:13:16+00:00', '32', '"xss csrf id rfe lfi"', '"REQUEST.name=%3Cscript%3Ealert%281%29%3C%2Fscr
ipt%3E GET.name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E"', '"%2Fvulnerabilities%2Fxss_r%2F%3Fname%3D%253Cscript%253Ealert%252
81%2529%253C%252Fscript%253E"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-15T09:13:16+00:00
Possible attack types "xss csrf id rfe lfi"
HTTP Requests "REQUEST.name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E GET.name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E"
Affected file/path: /vulnerabilities/xss_r/
Attack string: /vulnerabilities/xss_r/?name=%3Cscript%3Ealert%281%29%3C%2Fscript%3E
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
five
*******
*******
```

Figure 5.86: Reflected XSS detected by self-healing script in experiment 2 step 5.

The healing mechanism uses the output from phpcs-security-audit when inputting which files to heal, illustrated in figure 5.87. The healing mechanism is equivalent to that of section 5.4 scenario 1; `htmlspecialchars()` is applied to user input.



```
******
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/help/help.php
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/index.php
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/source/high.php
Applying sanitization using htmlspecialchar
Applying sanitization using htmlspecialchar
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/source/low.php
Applying sanitization using htmlspecialchar
Applying sanitization using htmlspecialchar
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/source/impossible.php
Applying sanitization using htmlspecialchar
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/xss_r/source/medium.php
Applying sanitization using htmlspecialchar
Applying sanitization using htmlspecialchar
******
```

Figure 5.87: Self-healing process in experiment 2 step 5.

For the reflected XSS, a comparison between code of file `low.php` before healing (listing 5.20) and after healing (listing 5.21) shows that the PHP sanitization function is implemented on user input, preventing harmful code injections.

```php
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>
```

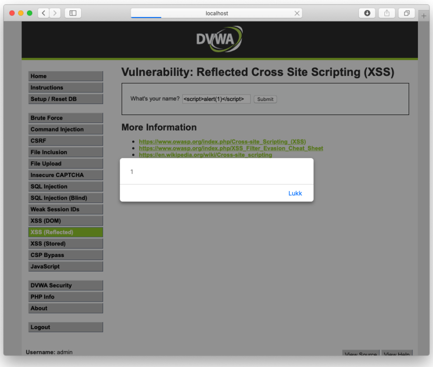
Listing 5.20: `low.php` before healing SQL injection in experiment 1 step 5.

```php
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && htmlspecialchars($_GET[ 'name'
    ]) != NULL ) {
```

```
    // Feedback for end user
    echo '<pre>Hello ' . htmlspecialchars($_GET[ 'name' ]) . '</pre>';
}

?>
```

Listing 5.21: `low.php` after healing SQL injection in experiment 1 step 5.

Figure 5.88 and 5.89 illustrates output from the browser before and after healing given the attack string. This clearly shows that the browser is prevented from running malicious scripts after being healed.



Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? <script>alert(1)</script>

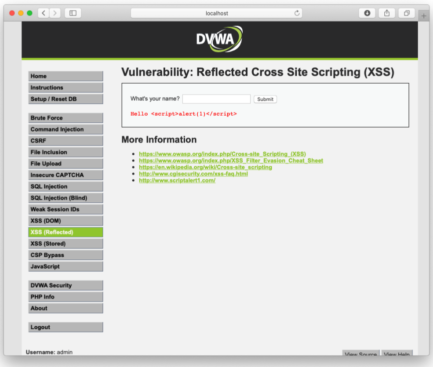
More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting

1

Username: admin

Figure 5.88: Browser output from reflected XSS attack before healing in experiment 1 step 5.



Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

```
Hello <script>alert(1)</script>
```

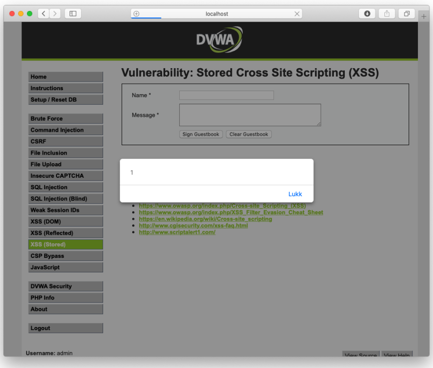
More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Username: admin

Figure 5.89: Browser output from reflected XSS attack after healing in experiment 1 step 5.

For the second example using stored XSS, the browser output before and after healing are presented in figure 5.90 and 5.91. The experiment yields the same results as reflected XSS; hence, the user input is sanitized.



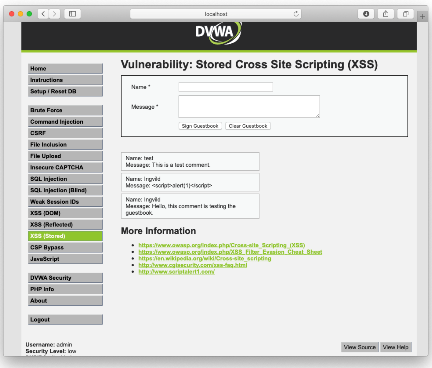
Vulnerability: Stored Cross Site Scripting (XSS)

1

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Username: admin

Figure 5.90: Browser output from stored XSS attack before healing in experiment 1 step 5.



Vulnerability: Stored Cross Site Scripting (XSS)

Name: test
Message: This is a test comment.

Name: Ingvild
Message: <script>alert(1)</script>

Name: Ingvild
Message: Hello, this comment is testing the guestbook.

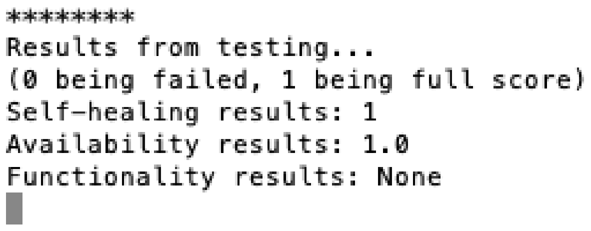
More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Username: admin
Security Level: low

Figure 5.91: Browser output from stored XSS attack after healing in experiment 1 step 5.

Results from the tests of the reflected XSS, which also are representative for the stored XSS, are presented in figure 5.92.

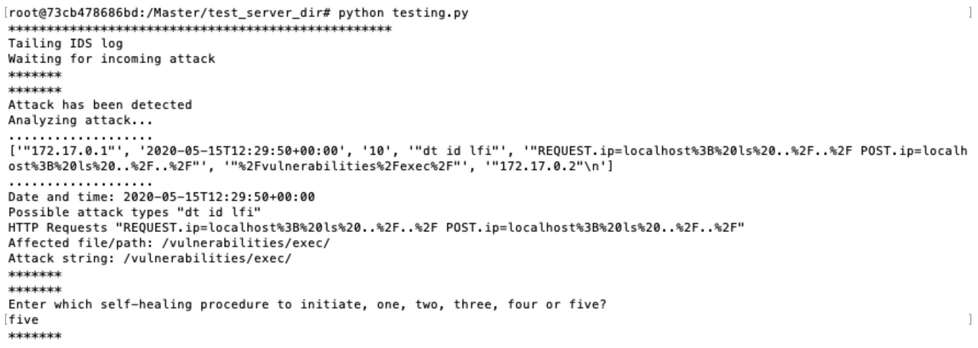


```
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: None
```

Figure 5.92: Results from experiment 2 step 5.

Experiment 3 - Command injection

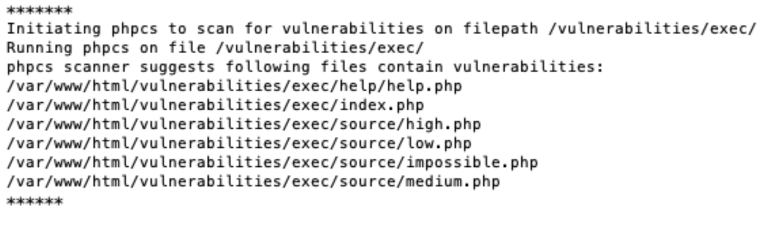
For the command injection, input **"localhost; ls ../../"** was used as attack input. This results in the IDS reacting and predicts the attack to be either **"dt id lfi"**, shown in figure 5.93.



```
root@73cb478686bd:/Master/test_server_dir# python testing.py
**************************************************
Tailing IDS log
Waiting for incoming attack
*******
*******
Attack has been detected
Analyzing attack...
...................
['"172.17.0.1"', '2020-05-15T12:29:50+00:00', '10', '"dt id lfi"', '"REQUEST.ip=localhost%3B%20ls%20..%2F..%2F POST.ip=localh
ost%3B%20ls%20..%2F..%2F"', '"%2Fvulnerabilities%2Fexec%2F"', '"172.17.0.2"\n']
...................
Date and time: 2020-05-15T12:29:50+00:00
Possible attack types "dt id lfi"
HTTP Requests "REQUEST.ip=localhost%3B%20ls%20..%2F..%2F POST.ip=localhost%3B%20ls%20..%2F..%2F"
Affected file/path: /vulnerabilities/exec/
Attack string: /vulnerabilities/exec/
*******
*******
Enter which self-healing procedure to initiate, one, two, three, four or five?
five
*******
```

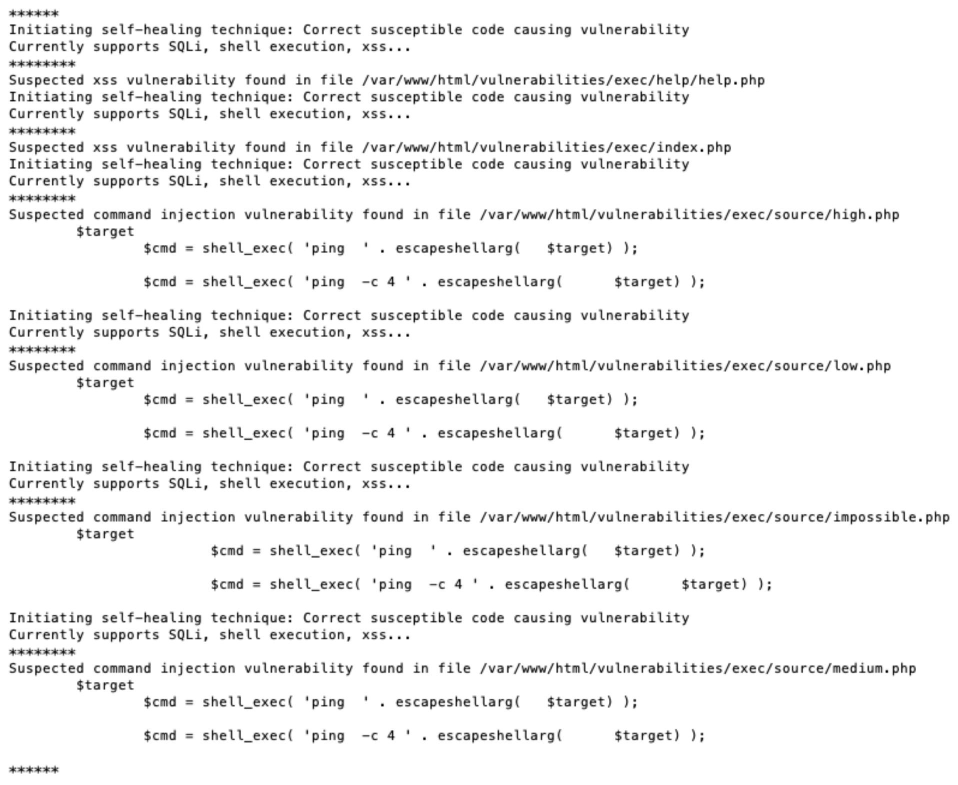
Figure 5.93: Self-healing script initiated in experiment 3 step 5.

The results from the vulnerability scanner (figure 5.94) is used in the healing process to decide which files to heal. The output from the self-healing is presented in figure 5.95, highlighting which lines are sanitized using `escapeshellarg()`.



```
*******
Initiating phpcs to scan for vulnerabilities on filepath /vulnerabilities/exec/
Running phpcs on file /vulnerabilities/exec/
phpcs scanner suggests following files contain vulnerabilities:
/var/www/html/vulnerabilities/exec/help/help.php
/var/www/html/vulnerabilities/exec/index.php
/var/www/html/vulnerabilities/exec/source/high.php
/var/www/html/vulnerabilities/exec/source/low.php
/var/www/html/vulnerabilities/exec/source/impossible.php
/var/www/html/vulnerabilities/exec/source/medium.php
******
```

Figure 5.94: phpcs-security-audit results in experiment 3 step 5.



```
******
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/exec/help/help.php
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected xss vulnerability found in file /var/www/html/vulnerabilities/exec/index.php
Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected command injection vulnerability found in file /var/www/html/vulnerabilities/exec/source/high.php
        $target
                $cmd = shell_exec( 'ping  ' . escapeshellarg(   $target) );

                $cmd = shell_exec( 'ping  -c 4 ' . escapeshellarg(      $target) );

Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected command injection vulnerability found in file /var/www/html/vulnerabilities/exec/source/low.php
        $target
                $cmd = shell_exec( 'ping  ' . escapeshellarg(   $target) );

                $cmd = shell_exec( 'ping  -c 4 ' . escapeshellarg(      $target) );

Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected command injection vulnerability found in file /var/www/html/vulnerabilities/exec/source/impossible.php
        $target
                        $cmd = shell_exec( 'ping  ' . escapeshellarg(   $target) );

                        $cmd = shell_exec( 'ping  -c 4 ' . escapeshellarg(      $target) );

Initiating self-healing technique: Correct susceptible code causing vulnerability
Currently supports SQLi, shell execution, xss...
********
Suspected command injection vulnerability found in file /var/www/html/vulnerabilities/exec/source/medium.php
        $target
                $cmd = shell_exec( 'ping  ' . escapeshellarg(   $target) );

                $cmd = shell_exec( 'ping  -c 4 ' . escapeshellarg(      $target) );

******
```

Figure 5.95: Self-healing process in experiment 3 step 5.

After healing, listing 5.22 and 5.23 illustrate the changed lines of the vulnerable files `low.php` before and after healing. The misconfiguration is that the function `shell_exec()` takes in direct user input, making it vulnerable to remote command executions from arbitrary users. As presented in listing 5.23, the user input is now sanitized before being executed.

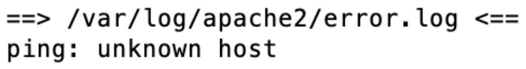
```php
<?php

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

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

Listing 5.22: `low.php` before healing command injection vulnerability in experiment 3 step 5.

```
<?php

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

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . escapeshellarg(    $target) );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . escapeshellarg(    $target)
    );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

Listing 5.23: `low.php` after healing command injection vulnerability in experiment 3 step 5.

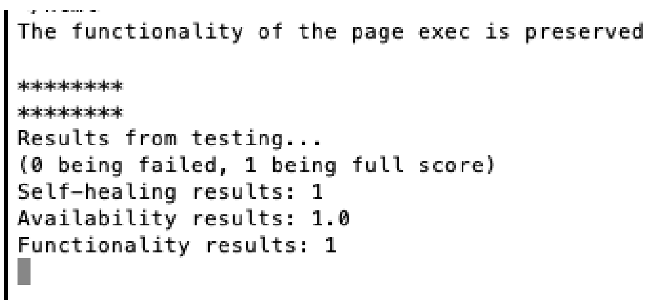
When testing if the vulnerability has healed by relaunching the attack, the attack is "rejected" by the server, leaving the browser blank (figure 5.96). The `/var/log/apache2/error.log` shows that the `ping` to be executed with the attack input "**localhost; ls ../../**" fails since the host does not exist. Using the sanitization function `escapeshellarg()` on the given attack input returns `'localhost; ls ../../'`. Hence, `ping 'localhost; ls ../../'` is executed instead, which is an unknown host.



```
==> /var/log/apache2/error.log <==
ping: unknown host
```

Figure 5.96: Ping fail when executing attack after healing in experiment 3 step 5.

The results from the three evaluation measurements are presented in figure 5.97. They all achieve full score on the scale.



```
The functionality of the page exec is preserved

********
********
Results from testing...
(0 being failed, 1 being full score)
Self-healing results: 1
Availability results: 1.0
Functionality results: 1
```

Figure 5.97: Results from experiment 3 step 5.

Summary of results

The self-healing technique removes the vulnerability, which is identified by patching coding errors. The three experiments conducted healing of common web vulnerabilities such as XSS and SQL injections. They were all able to heal the vulnerability while meeting the availability and functionality criteria. Hence, this healing technique performs the best out of all five with regards to the evaluation criteria. However, these results can only be applied to these special cases on this web server and is therefore not scaleable to other systems. The technique, as of now, only has support for the vulnerabilities tested in the experiment, when in reality, there are multitudes of other web vulnerabilities that might be present.

	Self-healing test	**Availability test**	**Functionality test**	**Average**
Exp. 1	1	1	1	1
Exp. 2	1	1	None	1
Exp. 3	1	1	1	1
Average	1	1	1	1

Table 5.6: Summary of results for step 5. The first three rows give the results from each experiments, including the average performance of each experiment in the final column. The last row gives the average results from all experiments, and the bottom right cell gives an idea of how well the self-healing technique performed concerning the evaluation criteria. The table shows that healing technique 5 showed significant performance in all the three experiments.

5.6 Chapter summary

In this chapter, a self-healing script was introduced, explained, and demonstrated with concrete cases. The script has one source of detecting breaches, namely an IDS. The script uses the detection mechanism to initiate either one of five self-healing procedures. The first self-healing mechanism performs healing by turning off the server whenever a security incident has occurred. The second process removes files which are indicated by the vulnerability scanner to contain vulnerabilities, while the third self-healing process removes the lines in the code of the susceptible files, which might contain vulnerabilities. The three first self-healing steps heal uniformly with regards to exploit, meaning that they heal the same way independent of what the vulnerability and attack parameters are. Step four and five use sanitization and patching to remove specific vulnerabilities, performing better on both availability and functionality criteria. However, in certain experiments of step 4, the healing criteria is not met since sanitization and filtering are used wrongly for the purpose of removing vulnerabilities. Table 5.7 presents the scores of all healing mechanisms based on the evaluation criteria.

The ideal healing mechanism for a system having been prone to a cyberattack, is to remove the vulnerability without disrupting other characteristics of the server, such as availability and functionality. However, the results from this testing prove that the intersection between guaranteed healing while preserving other essential traits is challenging. The more comprehensive and extreme measures to remove vulnerabilities, such as steps 1, 2, and 3, are less complex to implement and act in some ways as failsafe procedures. However, step 4, and especially step 5, providing better performance on all three evaluation criteria, require significantly more entangled and composite development and implementation, as is clearly demonstrated in the

	Average result from experiments
Healing technique 1	0.333
Healing technique 2	0.641
Healing technique 3	0.666
Healing technique 4	0.867
Healing technique 5	1

Table 5.7: Summary of results for all self-healing techniques. The table includes the average result from each healing technique in the rows. It clearly indicates that, according to the set evaluation criteria, step 1 performs the poorest, while step 5 has the best scores.

experiments. Nevertheless, all steps need unit and regression tests for functionality measures, which is also a time-consuming task to create for each accessible file. Nevertheless, these results are specific to the conducted experiments of this thesis and cannot necessarily be applied to all vulnerable servers.

Chapter 6
Discussion

In this chapter, the research questions of the thesis are discussed against the achieved results. Further, the arising limitations of the project will be discussed, as well as suggestions for future work on the topic.

6.1 RQ1: Pre-existing self-healing techniques

The first research question, *"What kind of self-healing techniques already exist, what are the shortcomings of these, and how can one overcome these?"*, are discussed and presented in the literature review of chapter 2. Briefly summarized, autonomous systems are a sought-after research topic. Self-healing is not a standardized term, hence research within the field varies in definitions. Many approaches to healing exist, such as machine learning, knowledge base consulting, exploitation of code redundancy, and systems inspired by cell's regenerative abilities. The most significant shortcoming that stands out is that systems claiming to be self-healing, has properties of fault-tolerance rather than healing.

6.2 RQ2: Patching and immunising vulnerabilities autonomously

The second research question, *"Which vulnerabilities that enable attacks will the system be able to patch autonomously, and which ones will it be able to immunize autonomously?"*, have been investigated through numerous experiments and tests. The testbed, DVWA, contained multiple web vulnerabilities, and several of these were tested in the practical experiments.

The five healing techniques achieved different results with regards to patching and immunizing. Several vulnerabilities were tested and successfully healed according to the set healing evaluation criteria, such as stored and reflected XSS, SQL injections, command injections, and local file inclusions. Concerning immunizing vulnerabilities, that is to neutralize or render them ineffective, steps 1, 2, 3, and partly 4, fall into this category. Ranging from turning off the vulnerable server to mitigating vulnerabilities by sanitizing user input, healing by immunizing was achieved.

Patching, namely to make corrections in the susceptible code to eliminate distinct vulnerabilities, was realized in steps 4 and 5. However, the two achieved different results; step 4 applied superficial, unspecific filtering and sanitization methods mainly focused on removing symptoms of the vulnerabilities. Step 5, on the other hand, targeted the root cause of the present vulnerabilities.

However, these experiments barely scratch the surface of existing web vulnerabilities. If the five healing steps were to be evaluated solely on coverage, step 1 would be the best option since it will block any attack due to the server being offline. It is also the most independent solution considering the use of external tools, since the only assumption is that the IDS is able to detect all breaches. The other healing techniques rely on the correctness of the vulnerability scanner as well. Step two is purely dependent on the vulnerability scanner to identify the correct, vulnerable files, whereas step 3 also needs the identification of vulnerable lines of code to be accurate. Steps 4 and 5 rely on both the scanner's and the IDS's verbose output to be able to heal the vulnerability correctly. Following this reasoning, step 1 is more probable to successfully immunize other vulnerabilities launched towards the server in addition to the ones conducted in the experiments, rather than the other steps since they require less drastic and more specific healing measures based on the exploit detected. In the same manner, step 2's coverage is probably greater than steps 3, 4, and 5, and so on.

6.3 RQ3: Correlation between self-healing techniques

In light of the third research question, *"If vulnerabilities are able to heal or be immunized, is there a correlation between the self-healing techniques?"*, comparisons between the functionality of the self-healing techniques will be made. Some of the methods provide more general, exhaustive solutions to healing of the vulnerabilities, namely step 1, 2, and 3. Step 4 and 5, on the other side, present solutions that target specific web vulnerabilities. The two approaches influence at least two corresponding consequences; the trade-offs between healing, availability and functionality and the complexity of the healing solutions.

For all healing steps and their corresponding successful experiments concerning healing, except for step 5, there is a trade-off between removal of exploitable segments, and availability and functionality. Step 1 is arguably the best option when healing is the primary object, but has proven to be severely inefficient with regards to availability and functionality. In the conducted experiments, step 2 sacrifices parts of the server availability and completely the functionality of the affected files. Step 3 achieved better results on availability, but in turn, the functionality is still lost. Steps 4 and 5 have the best test results from the experiment, minimizing the trade-off between the three requirements. However, the experiments of step 4 did not exclusively achieve successful healing.

The experiments with the five healing techniques indicate that rising complexity and a greater need for code awareness are outcomes from more specific functionality preserving healing techniques. The steps demonstrate the two outcomes by comparing the broader healing techniques with the more distinct and precise ones; for instance, both lines of code and code complexity increase due to an increase in handling specific cases, namely exploits.

6.4 Limitations

The results from the thesis might be affected by numerous factors, such as the choice of tools and platforms and how experiments were executed. In this section, we discuss potential limitations of the project.

6.4.1 Generalizability

The self-healing script, evaluation tests, and tools are customized and tailor-made for the chosen testbed. Therefore, the results are not directly applicable to other web servers or testbeds. For instance, the healing of the SQL injection in step 5 works as intended for the specific case on DVWA, but it is not generalized for all instances of SQL injections.

6.4.2 Evaluation execution and results

There are several potential limitations related to the evaluation criteria. Firstly, the evaluation of a self-healing technique's performance is based solely on three requirements; a substantial amount of requirements and criteria would provide a more comprehensive assessment. Secondly, the functionality tests are limited in number and complexity; there are only three functionality tests, and these are fairly simple. However, the functionality of the pages on DVWA is rather minimalistic, making it difficult to test how intricate systems would respond to the self-healing techniques. Thirdly, the self-healing test is not automated, and as one of the aims of self-healing systems is to act as autonomous beings, the credibility of the self-healing script would have improved if all tests were automated. Finally, the implementation of the availability test presented a weakness, especially for step 3, since it does not account for unavailability due to internal server errors (HTTP error response 500).

Even though the page is "up," it behaves similarly to the opposite, which could be beneficial to include in the availability results.

6.4.3 Vulnerability coverage and dependability on external tools

The solution is entirely dependent on an IDS for detecting cyberbreaches and on a vulnerability scanner for detecting vulnerabilities. Therefore, the solution can only be as good as the tools it relies on. When using the tools, it is clear that it is possible to bypass the IDS (such as a file upload vulnerability), and there exist vulnerabilities which the vulnerability scanner is unable to identify (such as a remote file inclusion vulnerability). For the latter, the specific vulnerability is undetected because the coding error is not in the PHP file, but in the configuration file `php.ini`. By using more diverse tools to detect vulnerabilities, the program could improve. For instance, the open-source tool *iniscan*[1] would have identified the remote file inclusion vulnerability and its patching could easily be automated.

6.4.4 Proof of Concept

Several of the self-healing techniques could have been applied before a breach occurred as multiple of the vulnerabilities are detectable without the contribution from an IDS. Therefore, it should be stated that the program mimics a proof of concept of a self-healing system, rather than having obtained essential characteristics of an entirely autonomous being.

6.5 Further work

In this thesis, we implemented a simple script to enforce self-healing capabilities on a web server. Experiments were conducted in a restricted environment with few tools and tests. To improve the coverage of the program, either more tools for the detection of breaches and identification of vulnerabilities can be integrated, or the ones chosen now can be revised and replaced with more suitable alternatives. Since the tests are both limited in number and test coverage, a better and more comprehensive testing scheme could be applied to achieve more realistic, credible, and precise results. Choosing another testbed with more intricate functionality could also provide new insights into the performance of the script. These improvements combined could potentially bring the self-healing program closer to the notion of being self-healing or an autonomous being.

[1]https://github.com/psecio/iniscan

Chapter 7 Conclusion

The development of self-healing computer systems is a hopeful contribution to the field of computer science with regards to effectiveness and maintenance, but perhaps most importantly, towards creating secure and safe systems with robust characteristics. In this thesis, the concept of self-healing was explored through a literature review and by developing a self-healing script monitoring a vulnerable web application.

RQ1: What kind of self-healing techniques already exist, what are the shortcomings of these and how can one overcome these In the literature study, several papers within the field of self-healing were examined. The techniques comprise of approaches such as machine learning, knowledge base consulting, exploitation of code redundancy, and systems inspired by biological procedures. It became apparent that the self-* terminology is used interchangeably, and there was not a clear definition of either one, as well as the research appears very limited. One of the most significant shortcomings identified is that the self-healing systems in certain works act more like a fault-tolerance system rather than a self-healing one. Referring to the classifications made of the related papers in the literature study, the technical work in this thesis covers implementation of a system that removes susceptibilities caused by security incidents.

RQ2: Which vulnerabilities that enable attacks will the system be able to patch autonomously, and which ones will it be able to immunize autonomously? During the technical testing of five self-healing techniques, multiple experiments were conducted by launching exploits towards a vulnerable web application. With the use of these techniques, several vulnerabilities, such as SQL injections and XSSs, were patched and immunized. However, during the experiments, only a handful of known PHP susceptibilities were investigated. The results indicated that the more radical healing solutions immunized a larger range of vulnerabilities than the specific healing solutions patched.

RQ3: If vulnerabilities are able to heal or be immunized, is there a correlation between the self-healing techniques? When comparing the self-healing techniques used, it became apparent that they preserved functional and non-functional requirements differently. Steps such as 1 and 2 achieved far worse results concerning maintaining availability and functionality after healing than step 5. However, as mentioned in the previous research question, it is anticipated that these low-scoring healing techniques will outperform the more precise healing techniques regarding healing if tested against several other vulnerabilities.

References

[15990] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[20112] *National cyber security framework manual.* 2012.

[20113] Iso / iec 25010 : 2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. 2013.

[Atk00] Leon Atkinson. *Core PHP programming: using PHP to build dynamic Web sites.* Pearson Education, 2000.

[BNS07] David Brumley, James Newsome, and Dawn Song. Sting: An end-to-end self-healing system for defending against internet worms. In *Malware Detection*, pages 147–170. Springer, 2007.

[Bru01] Guy Bruneau. The history and evolution of intrusion detection. *SANS Institute*, 1, 2001.

[CN95] Lawrence Chung and Brian A Nixon. Dealing with non-functional requirements: three experimental studies of a process-oriented approach. In *1995 17th International Conference on Software Engineering*, pages 25–25. IEEE, 1995.

[CRC08] Patricia Cronin, Frances Ryan, and Michael Coughlan. Undertaking a literature review: a step-by-step approach. *British journal of nursing*, 17(1):38–43, 2008.

[Dub13] Elena Dubrova. *Fault-tolerant design.* Springer, 2013.

[EA09] Muna Elsadig and Azween Abdullah. Biological inspired intrusion prevention and self-healing system for network security based on danger theory. *International Journal of Video & Image Processing and Network Security*, 9(9):16–28, 2009.

[Fre87] Peter Freeman. *Software perspectives: the system is the message.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[Gli07] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26. IEEE, 2007.

[GSRU07] Debanjan Ghosh, Raj Sharman, H Raghav Rao, and Shambhu Upadhyaya. Self-healing systems—survey and synthesis. *Decision support systems*, 42(4):2164–2185, 2007.

[Har18] Chris Hart. *Doing a literature review: Releasing the research imagination.* Sage, 2018.

[HFAAF17] Amjad A Hudaib, Hussam N Fakhouri, Fatima Eid Al Adwan, and Sandi N Fakhouri. A survey about self-healing systems (desktop and web application). *Communications and Network*, 9(1):71–88, 2017.

[How80] William E Howden. Functional program testing. *IEEE Transactions on Software Engineering*, (2):162–169, 1980.

[IHK04] Juhani Iivari, Rudy Hirschheim, and Heinz K Klein. Towards a distinctive body of knowledge for information systems experts: coding isd process knowledge in two is journals. *Information systems journal*, 14(4):313–342, 2004.

[Jav] Yasir Javed. Using public vulnerabilities data to self-heal security issues in software systems.

[JM19] Linda Joseph and Rajeswari Mukesh. Securing and self recovery of virtual machines in cloud with an autonomic approach using snapshots. *Mobile Networks and Applications*, 24(4):1240–1248, 2019.

[JNS16] David Jaramillo, Duy V Nguyen, and Robert Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5. IEEE, 2016.

[KB15] Joxean Koret and Elias Bachaalany. *The antivirus hacker's handbook.* Wiley Online Library, 2015.

[KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[Ker07] Angelos D Keromytis. Characterizing self-healing software systems. 2007.

[Kot04] Chakravanti Rajagopalachari Kothari. *Research methodology: Methods and techniques.* New Age International, 2004.

[LC09] Simon Liu and Bruce Cheng. Cyberattacks: Why, what, who, and how. *IT professional*, 11(3):14–21, 2009.

[LE06] Yair Levy and Timothy J Ellis. A systems approach to conduct an effective literature review in support of information systems research. *Informing Science*, 9, 2006.

[Lim06] Keng Leng Lim. Intrusion protection system and method, February 2 2006. US Patent App. 11/051,795.

[LW89] Hareton KN Leung and Lee White. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*, pages 60–69. IEEE, 1989.

[LW90] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301. IEEE, 1990.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on software engineering*, (6):483–497, 1992.

[MGK19] Anders Mikkelsen, Tor-Morten Grønli, and Rick Kazman. Immutable infrastructure calls for immutable architecture. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[MROO20] Hannah Ritchie Max Roser and Esteban Ortiz-Ospina. Internet. *Our World in Data*, 2020. https://ourworldindata.org/internet.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[Nas18] Nasjonal Sikkerhetsmyndighet. Grunnprinsipper for ikt-sikkerhet, versjon 1.1. 2018.

[OWA17] Top OWASP. Top 10-2017 the ten most critical web application security risks. *URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf*, 29, 2017.

[Per13] Nicolo Perino. A framework for self-healing software systems. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1397–1400. IEEE, 2013.

[PH04] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In *International workshop on unconventional programming paradigms*, pages 257–269. Springer, 2004.

[PK04] Barbara Paech and Daniel Kerkow. Non-functional requirements engineering-quality is essential. In *10th International Workshop on Requirments Engineering Foundation for Software Quality*, 2004.

[PKL$^+$09] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, 2009.

[Row02] Craig H Rowland. Intrusion detection system, June 11 2002. US Patent 6,405,318.

[RRS11] S Ramamoorthy, SP Rajagopalan, and S Sathyalakshmi. Process for security in self-healing systems' architecture. 2011.

[She08] Onn Shehory. Shadows: Self-healing complex software systems. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 71–76. IEEE, 2008.

Chapter A
Appendix

A.1 Self-healing script

```
import subprocess
import os
import urllib
import requests
import re
from sh import tail
import time

def watch(fn):
    fp = open(fn, 'r')
    while True:
        new = fp.readline()
        # Once all lines are read this just returns ''
        # until the file changes and a new line appears
  if new == '':
    print "**************************************************"
              print("Tailing IDS log")
    print("Waiting for incoming attack")
    print("*******")
    print("*******")
    while True:
      attack = fp.readline()
      if (attack != ""):
        print "Attack has been detected"
        print "Analyzing attack..."
        print "..................."
        p = attack.split(",")
        print p
        print "..................."
        ip1 = p[0]
        date= p[1]
        impact = p[2]
        type = p[3]
    reqget = p[4]
    req, get_post = p[4].split(" ")
    if "POST." in get_post:
      get_post = "POST"
    elif "GET." in get_post:
      get_post = "GET"
    subURL = getFilepath(p[5])[0]
    subURL = subURL.strip('"')
    attack_URL = getFilepath(p[5])[1]
    attack_URL = attack_URL.strip('"')
    ip2 = p[6]
    print "Date and time: "+date
    print "Possible attack types "+type
    print "HTTP Requests "+reqget
    print "Affected file/path: "+subURL #/vulnerabilities/xss_r/
    print "Attack string: "+attack_URL #/vulnerabilities/xss_r/?name=%3Cscript%3Ealert%281%29%3C%2
Fscript%3E
    print "*******"
    print "*******"
    ws = raw_input("Enter which self-healing procedure to initiate, one, two, three, four or five? \n")
    while ws not in ["one","two","three","four","five"]:
      ws = input("Enter which self-healing procedure to initiate, 1, 2, 3, 4, 5?")

    sh_res=""
    av_res=""
    func_res=""
    print "*******"
    print "*******"
                            print "Preparing availability criteria by checking availability of pages
before initiating healing."
    print "......."
    os.system("./get_HTTP_resp_before.sh")

    if  ws == "one":
          step1()
          status = av_test(subURL)
          sh = sh_test(subURL, p, fn,status)
          if sh != None:
            print "*********"
            print "Relaunch exploit manually with given attack url parameters"
            print " to validate if vulnerability is unreachable/healed:"
            print "URL: "+sh[0]
            print "Input: "+sh[2]
            print "*********"
            heal_noheal = raw_input("Press y or yes to confirm healing \n")
            if heal_noheal == "y" or heal_noheal == "yes":
              sh_res = 1
              func_res = 0
              av_res = 0
          if status != "down":
            print "Running func test"
            func_test(subURL)

          i = raw_input("Press y or yes to restart server \n")
                print("")
                if i == "y" or  i == "yes":
                        os.system("./restart_server.sh")
        else:

          phpcs_txt = phpcs(subURL,date)
          files_loc = phpcs_file_loc(phpcs_txt)
          all_files = []
          print "Running phpcs on file "+subURL
          print "phpcs scanner suggests following files contain vulnerabilities:"
          for f in files_loc:
            all_files.append(f[0])
            print f[0]
          if  ws == "two":
```

```
                                          step2(subURL,all_files)
            av = av_test(subURL)
            if not isinstance(av,basestring):
              num = av[0]
              after = av[2]
              av_res = (float(float(num)-float(after)))/float(num)
                                            status = ""

            sh = sh_test(subURL, p, fn,status)
            if sh != None:
                                                  print "*********"
                                                  print "Relaunch exploit manually with given attack url
    and parameters to validate if vulnerability is unreachable/healed:"
                                                  print "URL: "+sh[0]
                                                  print "Input: "+sh[2]
                                                  print "*********"
                                                  heal_noheal = raw_input("Press y or yes to confirm
    healing \n")
                                                  if heal_noheal == "y" or heal_noheal == "yes":
                sh_res = 1

            func_res = func_test(subURL)
          if  ws == "three":
            for entry in files_loc:
              step3(entry[0],entry[1])

            av = av_test(subURL)
                                                if not isinstance(av,basestring):
                                                        num = av[0]
                                                        after = av[2]
                                                        av_res = (float(float(num)-float(after)))/float(num
    )
                                                        status = ""
                                          sh = sh_test(subURL, p, fn, status)
```

```
                   if sh != None:
                          print "*********"
                          print "Relaunch exploit manually with given attack
url and parameters to validate if vulnerability is unreachable/healed:"
                          print "URL: "+sh[0]
                          print "Input: "+sh[2]
                          print "*********"
                          heal_noheal = raw_input("Press y or yes to confirm
healing \n")
                          if heal_noheal == "y" or heal_noheal == "yes":
                              sh_res = 1
     else:
      sh_res = 0

    func_res = func_test(subURL)

   if  ws == "four":

       attack_var_and_input_raw = req[len("REQUEST")+2:]
       attack_var, attack_input_raw = req.split("=")
       attack_var = attack_var[len("REQUEST")+2:]
                   for entry in files_loc:
     step4(type.split(" "), get_post, entry[0],entry[1], attack_var, attack_URL,entry[2], p[4])

    av = av_test(subURL)
                      if not isinstance(av,basestring):
                          num = av[0]
                          after = av[2]
                          av_res = (float(float(num)-float(after)))/float(num
)
                          status = ""

    sh = sh_test(subURL, p, fn, status)
                      if sh != None:
```

```
                                                    print "*********"
                                                    print "Relaunch exploit manually with given attack url and parameter to validate if vulnerability is unreachable/healed:"
                                                    print "URL: "+sh[0]
                                                    print "Input: "+sh[2]
                                                    print "*********"
                                                    heal_noheal = raw_input("Press y or yes to confirm healing \n")
                                                    if heal_noheal == "y" or heal_noheal == "yes":
                                                            sh_res = 1
                                                    else:
                                                            sh_res = 0

                                            func_res = func_test(subURL)

      if  ws == "five":
        print "******"
        attack_var_and_input_raw = req[len("REQUEST")+2:]
                                            attack_var, attack_input_raw = req.split("=")
                                            attack_var = attack_var[len("REQUEST")+2:]
                                            for entry in files_loc:
                                                    step5(type.split(" "), get_post, entry[0],entry[1], attack_var, entry[2])

                                            av = av_test(subURL)
                                            if not isinstance(av,basestring):
                                                    num = av[0]
                                                    after = av[2]
                                                    av_res = (float(float(num)-float(after)))/float(num)
                                                    status = ""

                                            sh = sh_test(subURL, p, fn, status)
                                            if sh != None:
```

```
                                                        print "*********"
                                                        print "Relaunch exploit manually with given attack
    url and parameter to validate if vulnerability is unreachable/healed: "
                                                        print "URL: "+sh[0]
                                                        print "Input: "+sh[2]
                                                        print "*********"
                                                        heal_noheal = raw_input("Press y or yes to confirm
    healing \n")
                                                        if heal_noheal == "y" or heal_noheal == "yes":
                                                                sh_res = 1
                                                        else:
                                                                sh_res = 0
            func_res = func_test(subURL)
        print ""
        print "********"
                                print "********"
                                print "Results from testing..."
        print "(0 being failed, 1 being full score)"
                                print "Self-healing results: "+ str(sh_res)
                                print "Availability results: "+ str(av_res)
                                print "Functionality results: "+ str(func_res)



def phpcs(filepath,date):
  print "*******"
  print "*******"
  print("Initiating phpcs to scan for vulnerabilities on filepath "+filepath)

  command = "php PHP_CodeSniffer/bin/phpcs --extensions=php,inc,lib,module,info --standard=/Master/
    test_server_dir/phpcs-security-audit/example_drupal7_ruleset.xml /var/www/html"+filepath
  command_list =  command.split(" ")
  filepath_output = "/Master/test_server_dir/phpcs_outputs/" #EDIT PATH
  filename = filepath[len("vulnerabilities")+1:]+"_"+date[:19]+".txt"
  filename = filename.replace("/", "")
  f = open(filepath_output+filename,"w")

  subprocess.call(command_list, stdout=f)

  return filepath_output+filename

def phpcs_file_loc(phpcs_results_txt):
    fp = open(phpcs_results_txt, 'r')
  files_loc=[]
  while True:
    line = fp.readline()
                locs_found=[]
                full_error_message = []

    if line[0:4] == "FILE":
      vuln_file = line[6:].rstrip("\n")
      while True:
        locs=fp.readline()
        if len(locs) > 1:
          if locs[1].isdigit() == True or locs[2].isdigit() == True or locs[3].isdigit() == True:
            if int(filter(str.isdigit, locs[0:4])) not in locs_found:
              locs_found.append(int(filter(str.isdigit, locs[0:4])))
                                      full_error_message.append(locs)
              continue
        elif len(locs) == 1:
          files_loc.append([vuln_file,locs_found,full_error_message])
          vuln_file =""
          locs_found=[]
          full_error_message=[]
          break
    if line[0:5] == "Time:":
      break
  fp.close()
  return files_loc

def getFilepath(badpath):
  #returns
  #('vulnerabilities/fi/', 'vulnerabilities/fi/?page=../../../../ls')
  pretty_URL = urllib.unquote(badpath)
  url = pretty_URL.split("?")[0]
  url = url[1:]
  return url, pretty_URL[1:len(pretty_URL)]

def step1():
  print("*******")
  print("Self-healing mechanism: turn of server")
  print("Server will shutdown now")
  #launch stop server script
  os.system("./stop_server.sh")
  print("*******")

def step2(filepath,vuln_files):
  #vulnfiles = all_files = filepath+name of all vulnerable files
  #get results from phpcs
  print("*******")
  print("Self-healing mechanism: remove vulnerable PHP file")
  print("*******")
  print("When scanning "+filepath+", phpcs suggests following files are vulnerable: "+ ', '.join(vuln_files
    ))
  print("*******")
  print("Moving file "+', '.join(vuln_files)+" from server to temporary/new folder")
  for file in vuln_files:
    #create functionality for blank file?
    os.system("mv "+file+" /Master/test_server_dir/files_from_step2/")
    print "* "+file+" is moved to /files_from_step2"


```

```
def step3(filepath, loc):
  #get results from phpcs
  #remove line of code in given php file, comment out
  print("*******")
  print("*******")
  print("Self-healing mechanism: remove vulnerable line of code")
  print("*******")
  f = open(filepath, "r")
  lines = f.readlines()
  f.close()
  print("Creating .old file for file "+filepath)
  name = filepath[len("/var/www/html/vulnerabilities/"):]
  name = name.replace("/","_")
  os.system("mv "+filepath+" /Master/test_server_dir/files_from_step3/"+name+".old")
  print("Copying lines of code to old file to path oldfiles/")
  mod_lines = lines
  print("Modifying original file's vulnerable lines of code")
  for l in loc:
    mod_line = "//"+lines[l-1]
    mod_lines[l-1] = mod_line
  print "The lines with modifications are: \n"
  for mline in mod_lines:
    print mline
  print("Overwriting original file with new, modified lines")
  nw = open(filepath, "w")
  nw.writelines(mod_lines)
  nw.close()
  print "*******"
  print "*******"

def step4(phpids_type, postget,filepath, loc, input_param,attack_url,phpcs_error_messages, attack_string):
  print"*****"
      print "Initiating self-healing technique: Correct susceptible code causing vulnerability"
      print "*******"
  print "Sanitizing user input based on phpids results and phpcs-security-audit results."

for i in range(len(phpids_type)):
 phpids_type[i] = phpids_type[i].strip('"')

if "xss" in phpids_type and (postget and "User input") in '\t'.join(phpcs_error_messages) and ("
 mysqli_query") not in '\t'.join(phpcs_error_messages):
    print "PHPIDS indicates XSS, while phpcs scanner detects user input with "+postget
    print "Initiating generic sanitization appropriate for possible xss: htmlspecialchars()"
    xss_heal(input_param, loc, filepath, postget)

  elif "sqli" in phpids_type and ("mysqli_query" and "dynamic parameter") in '\t'.join(
 phpcs_error_messages):
    print "PHPIDS indicates SQL injection, while phpcs scanner detects MYSQLi query with dynamic
 parameter "
    print "Initiating generic sanitization appropriate for possible sqli: mysqli_real_escape_string()
 "
 print "File: "+ filepath
    file = open(filepath, "r")
    lines = file.readlines()
    file.close()
    c = 0
 db_conn_var = '$GLOBALS["___mysqli_ston"]'
    for l in lines:
       if "$_[" and "'"+input_param+"'" in l and "isset" not in l:
         print "Finding index in string of $_"
         index_dollar = find_all_indexes(l,"$_")
         print "Finding closing bracket"
         for x in index_dollar:
            index_right_bracket = l[int(x):].find("]")
            index_right_bracket = x + index_right_bracket

            if input_param in l[x:index_right_bracket]:
              apply_san = l[x:index_right_bracket+1]
```

```
                          print "Applying sanitization using mysqli_real_escape_string()"
                          san = "mysqli_real_escape_string("+db_conn_var + "," +apply_san+")"
                          print "Sanitized line:"
        sanitized_line = l.replace(apply_san,san)
                          print sanitized_line
        lines[c] = sanitized_line
            c=c+1

      file2 = open(filepath, "w")
      file2.writelines(lines)
      file2.close()

  elif ("dt" in phpids_type and ("id" in phpids_type or "lfi" in phpids_type)) and "../../" in urllib.
unquote(attack_string):
      print "PHPIDS indicates attacks such as directory traversal, local file inclusion and command
injection."
      print "Initiating quick fix solution to avoid directory traversal and file inclusion: str_replace
()"
print "File: " + filepath
      file = open(filepath, "r")
      lines = file.readlines()
      file.close()
      c = 0
      for l in lines:
            if "$_[" and "'" + input_param + "'" in l:
                print "Finding index in string of $_"
                index_dollar = find_all_indexes(l, "$_")
                print "Finding closing bracket"
                for x in index_dollar:
                      index_right_bracket = l[int(x):].find("]")
                      index_right_bracket = x + index_right_bracket

                      if input_param in l[x:index_right_bracket]:
                          apply_san = l[x:index_right_bracket + 1]
```

```
             print "Applying sanitation using str_replace()"
             san = 'str_replace( array( '+'"'+'../'+'"'+"), "+'"", '+ apply_san + ")"
             print san
             sanitized_line = l.replace(apply_san, san)
             lines[c] = sanitized_line
 c = c + 1
file2 = open(filepath, "w")
   file2.writelines(lines)
   file2.close()

 else:
   print "System does not support anymore speicific sanitization options towards certain
vulnerabilities"
   print "Basic sanitization will be applied to file "+filepath
file = open(filepath, "r")
   lines = file.readlines()
   file.close()
   c = 0
   for l in lines:
      if "$_[" and "'" + input_param + "'" in l:
        print "Finding index in string of $_"
        index_dollar = find_all_indexes(l, "$_")
        print "Finding closing bracket"
        for x in index_dollar:
           index_right_bracket = l[int(x):].find("]")
           index_right_bracket = x + index_right_bracket

           if input_param in l[x:index_right_bracket]:
             apply_san = l[x:index_right_bracket + 1]
             print "Applying sanitization using preg_replace()"
             san = "preg_replace("+"'"+"/[^a-zA-Z0-9.]/"+"'"+", "+"''"+""+", " + apply_san
 + ")"
             print san
             sanitized_line = l.replace(apply_san, san)
```

```python
                              lines[c] = sanitized_line
                c = c + 1
              file2 = open(filepath, "w")
              file2.writelines(lines)
              file2.close()


def step5(phpids_type,postget, filepath, loc, input_param,phpcs_error_messages):
  print "Initiating self-healing technique: Correct susceptible code causing vulnerability"
  print "Currently supports SQLi, shell execution, xss..."
  print "********"

  for i in range(len(phpids_type)):
                phpids_type[i] = phpids_type[i].strip('"')

  if "sqli" in phpids_type and "MYSQLi" in '\t'.join(phpcs_error_messages):
    sqli_heal(input_param,'$GLOBALS["___mysqli_ston"]',filepath)
  elif "shell_exec" in '\t'.join(phpcs_error_messages) and ("dt" in phpids_type or "id" in phpids_type or "lfi" in phpids_type):
    exec_heal(input_param, loc, filepath)
  elif ("User input" and postget) or "HTML construction with direct user input $_GET detected" in '\t'.join(phpcs_error_messages) and "xss" in phpids_type:
    xss_heal(input_param, loc, filepath, postget)

def find_all_indexes(input_str, search_str):
    l1 = []
    length = len(input_str)
    index = 0
    while index < length:
        i = input_str.find(search_str, index)
        if i == -1:
            return l1
        l1.append(i)
        index = i + 1
    return l1 #https://www.journaldev.com/23666/python-string-find


def xss_heal(input_params,loc, filepath,getpost):
    print("Suspected xss vulnerability found in file " + filepath)
    file = open(filepath, "r")
    lines = file.readlines()
    file.close()
    var_name_i_p = ""

    for input_param in input_params:

        for l in loc:
            if "$_"+getpost in lines[int(l)-1] and "htmlspecialchars" not in lines[int(l)-1] and "isset"
    not in lines[int(l)-1]:
                #print "Finding index in string of $_"+getpost+"["
                index_dollar = find_all_indexes(lines[int(l)-1],"$_"+getpost+"[")
                #print "Finding closing bracket"
                for x in index_dollar:

                    index_right_bracket = lines[int(l)-1][int(x):].find("]")
                    index_right_bracket = x + index_right_bracket

                    if input_param in lines[int(l)-1][x:index_right_bracket]:
                        apply_san = lines[int(l)-1][x:index_right_bracket+1]
                        print "Applying sanitization using htmlspecialchar"
                        san = "htmlspecialchars("+apply_san+")"
                        sanitized_line = lines[int(l)-1].replace(apply_san,san)
                        lines[int(l)-1] = sanitized_line

    os.system("cp "+filepath+" "+filepath+".old")
    file2 = open(filepath, "w")
    file2.writelines(lines)
    file2.close()
```

```


def exec_heal(input_param,loc, filepath):
    print("Suspected command injection vulnerability found in file " + filepath)

    file = open(filepath, "r")
    lines = file.readlines()
    file.close()
    var_name_i_p = ""

    for i in range(len(lines)):
        if "'"+input_param+"'" in lines[i]:
            var_name_i_p = lines[i].split("=")[0].strip(" ")
            print var_name_i_p

    for l in loc:
 if "shell_exec" in lines[l-1]:
     new_line = lines[int(l)-1].replace(var_name_i_p.replace("\t", ''),"escapeshellarg("+var_name_i_p+
  ")")
     print new_line
     lines[int(l)-1] = new_line


    os.system("cp "+filepath+" "+filepath+".old")
    file2 = open(filepath, "w")
    file2.writelines(lines)
    file2.close()


def sqli_heal(input_param, db_conn_var, filepath):
    #'$GLOBALS["___mysqli_ston"]'
    print("Suspected SQLi vulnerability found in file "+ filepath)

    file = open(filepath, "r")
    lines = file.readlines()
    file.close()
    changed_lines = []
    var_name_i_p=""

    for i in range(len(lines)):
        if "'"+input_param+"'" in lines[i]:
            var_name_i_p = lines[i].split("=")[0].strip(" ")

        if var_name_i_p and ("SELECT" or "FROM" or "WHERE" or "ORDER_BY") in lines[i]:
            query_found = lines[i]
            var2 = var_name_i_p.replace("\t", "")
            new_query = lines[i].replace("'"+var2+"';","?") #beware ;
            index = i
            for x in range(index):
                changed_lines.append(lines[x])
            var_name_qry = new_query.split("=")[0].strip(" ")
            changed_lines.append(new_query)

            prep = "    $prep = mysqli_prepare("+db_conn_var+","+var_name_qry+"); \n"
            var_name_prep = "$prep"
            changed_lines.append(prep)

            bind = "    mysqli_stmt_bind_param("+var_name_prep+","+"'s'"+","+var_name_i_p+"); \n"
            changed_lines.append(bind)

            execute = "    mysqli_stmt_execute("+var_name_prep+"); \n"
            changed_lines.append(execute)

        elif "mysqli_query" in lines[i]:
            result_found = lines[i]
            var_name_result = lines[i].split("=")[0].strip(" ")
            result = result_found.split("=")[0]+"="+"mysqli_stmt_get_result("+var_name_prep+"); \n"
            changed_lines.append(result)

            for c in range(i+1, len(lines)):
                changed_lines.append(lines[c])
            break

    file2 = open(filepath, "w")
    file2.writelines(changed_lines)
    file2.close()
    print "SQL injection fixed."

#The following function is an attempt of automated healing testing, but is not functioning correctly.
def sh_test(attacked_page,p,fn, status):
  print ""
  print "*******"
  print("Initiating self-healing test for self-healing technique ...")

  req, get_post = p[4].split(" ")

  if "POST." in get_post:
    get_post = "POST"
  elif "GET." in get_post:
    get_post = "GET"


  attack_var_and_input_raw = req[len("REQUEST")+2:]
  attack_var, attack_input_raw = req.split("=")
  attack_var = attack_var[len("REQUEST")+2:]
  attack_input_unquote = urllib.unquote(attack_input_raw)
  attack_url = "http://localhost"+attacked_page

  data = {attack_var : attack_input_unquote}
  headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like
     Gecko) Chrome/39.0.2171.95 Safari/537.36'}
  payload = {
    'username': 'admin',
    'password': 'password',
    'Login': 'Login'
}
  c2 = {'PHPSESSID': '8fksgel1eq8fhq6hfnhabbtuf1', 'security':'low'}

  cookies={}
  url1 = attack_url
  url2 = attack_url+"?"+attack_var+"="+attack_input_raw

  #if status == "down":
    #return url1, url2, attack_input_unquote
  return url1, url2, attack_input_unquote

  with requests.Session() as c:
        r = c.get('http://localhost/login.php',headers=headers)
        for e in r.cookies:
            cookies[e.name] = e.value
        token = re.search("user_token'\s*value='(.*?)'", r.text).group(1)
        payload['user_token'] = token
        p = c.post('http://localhost/login.php', data=payload,headers=headers)
  burp0_cookies = { "security": "low", "PHPSESSID": "fg402r9lvp39hugen8ve89lgg3"}
  session = requests.Session()
  print get_post
  if get_post == "POST":
    print "POST"
    test = session.post(url1,headers=headers,cookies=burp0_cookies,data=data)
    o = session.post(url2,headers=headers,cookies=burp0_cookies)
  else:
    print "in get place too"
    c.get(url1,headers=headers,cookies=burp0_cookies,data=data)
                c.get(url2,headers=headers,cookies=burp0_cookies)
```

```
def rem_302(file):
  f = open(file,"r")
  list = f.readlines()
  for l in list:
    if "302" in l:
      list.remove(l)
  f.close()
  n = open(file,"w")
  n.writelines(list)
  n.close()

def av_test(files):
  print "******"
  print("Initiating avaibability test after self-healing technique ...")
  path_to_pages = "/Master/test_server_dir/path_to_files.txt"
  print("Checking availability of files on the web server.")
  os.system("./get_HTTP_resp.sh")
  rem_302("avail_crit/output_http_resp.txt")
        rem_302("avail_crit/http_resp_before_heal.txt")
  print("******")
  print("******")
  print("Comparing up and down sites before and after self-healing...")
  print("******")
  print("******")
  before = open("avail_crit/http_resp_before_heal.txt","r")
  after = open("avail_crit/output_http_resp.txt","r")
  before_lines = before.readlines()
  after_lines = after.readlines()
  before.close()
  after.close()

  num = (len(before_lines)/2)
  after_404 = 0
  before_404 = 0
  file_before_404 = []
  file_after_404 = []
  for i in range(len(before_lines)): #assumes no file with name including 404
    if "404\n" in  before_lines[i]:
      before_404 = before_404 + 1
      file_before_404.append(before_lines[i-1])
  for y in range(len(after_lines)):
    if "404\n" in  after_lines[y]:
                        after_404 = after_404 + 1
                        file_after_404.append(after_lines[y-1])

  if after_404 != 0 or before_404 != 0 or len(after_lines)==num:
    if before_404 > 0:
      print("The pages down before healing:")
      for r in file_before_404:
        print r
    if after_404 > 0 and len(after_lines) != 0:
                        print("The pages down after healing:")
                        for k in file_after_404:
                                print k
      print("Out of "+str(num)+" pages, before:"+str(before_404)+" and after:"+str(after_404)+" were unavailable")

    if len(after_lines) == num:
      print("All pages, "+str(num)+" are down after healing >> The host is down")
      return "down"
  else:
    print("All pages are up, total: "+ str(num))

  return num, before_404, after_404
#av_test(1)

def authenticate_to_DVWA():

  print("Running authenticating procedure to server...")

        headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36'}
        payload = {
    'username': 'admin',
    'password': 'password',
    'Login': 'Login'
}
        cookies={}

        with requests.Session() as c:
                r = c.get('http://localhost/login.php',headers=headers)
                for e in r.cookies:
                        cookies[e.name] = e.value
                token = re.search("user_token'\s*value='(.*?)'", r.text).group(1)
                payload['user_token'] = token
                p = c.post('http://localhost/login.php', data=payload,headers=headers)
        burp0_cookies = { "security": "low", "PHPSESSID": "fg402r9lvp39hugen8ve89lgg3"}
        session = requests.Session()
  print("... Authenticated")
  return session, burp0_cookies, headers

def func_test(file_to_test):
  print("")
  print("*******")
  print("Initiating functionality test for file "+file_to_test)
  print("Currently supports unit/regression tests for /exec/, /fi/ and /sqli/")
  print("******")

  ci_expected = ""
  sql_expected = ""
  fi = ""

```

```
session, cookies, headers = authenticate_to_DVWA()

if file_to_test == "/vulnerabilities/exec/":
  print "For /vulnerabilities/exec, the expected behavior is to ping given ip or url."
  url = "http://localhost"+file_to_test
  data = {"ip":"localhost", "Submit":"submit"}
  exec_test = session.post(url, headers=headers, cookies=cookies, data=data)
  print exec_test.text
  if "0 packets received" not in exec_test.text and exec_test.text != "":
    print "The functionality of the page exec is preserved"
    return 1
  else:
    print "The functionality has been distorted somehow. The output is not as expected when trying to ping. See output:"
    print ""
    print exec_test.text
    return 0

if file_to_test == "/vulnerabilities/fi/":
  print "For "+file_to_test+", the expected behavior is to display either one of three files. These must be accessible."
  base_url = "http://localhost"+file_to_test+"?page="
  files = ["file1.php","file2.php","file3.php"]
  c = float(0)
  print "The three files are tested one by one."

  fi_test_1 = session.get(base_url+files[0],headers=headers,cookies=cookies)
  #can do even more extensive tests for each file, ex for this one check if the IP address presented is the same as mine
  if "File 1" and "Hello" and "Your IP address is" in fi_test_1.text:
    print "The functionality of the file file1.php in functionality test of /fi/ has been preserved."
    c = c + float(1)/float(3)
  else:
    print "The functionality of file1.php is not preserved."
```

```

 fi_test_2 = session.get(base_url+files[1],headers=headers,cookies=cookies)
       if "I needed a password eight characters long so I picked Snow White and the Seven Dwarves" in fi_test_2.text:
           print "The functionality of the file file2.php in functionality test of /fi/ has been preserved."
        c = c + float(1)/float(3)
 else:
           print "The functionality of file2.php is not preserved."

 fi_test_3 = session.get(base_url+files[2],headers=headers,cookies=cookies)
       #can do even more extensive tests for each file, ex for this one check if the IP address presented is the same as mine
       if "File 3" and "Welcome back" and "Your user-agent address is" in fi_test_3.text:
           print "The functionality of the file file3.php in functionality test of /fi/ has been preserved."
        c = c + float(1)/float(3)
 else:
           print "The functionality of file3.php is not preserved."

 return c

if file_to_test == "/vulnerabilities/sqli/":
 print "For "+file_to_test+", the expected behavior is to display ID, first name and surname when given an ID as input."
 url = "http://localhost"+file_to_test+"?id=1&Submit=Submit#"
 sqli_test = session.get(url, headers=headers, cookies=cookies)
 if "<pre>ID: 1<br />First name: admin<br />Surname: admin</pre>" in sqli_test.text:
  print "<pre>ID: 1<br />First name: admin<br />Surname: admin</pre> found in output."
  print "The functionality of /sqli/ has been preserved after healing."
  return 1
 else:
  print "The functionality of /sqli/ has not been preserved after healing."
  print "See output:"
```

```
      print ""
      print sqli_test.text
      return 0
  else:
    print "There is not support to test this filepath: "+file_to_test

fn = '/var/www/html/external/phpids/0.6/lib/IDS/tmp/phpids_log.txt'

watch(fn)
```

A.2 Example output from phpcs-security-audit

```

FILE: /var/www/html/vulnerabilities/exec/help/help.php
----------------------------------------------------------------------
FOUND 0 ERRORS AND 2 WARNINGS AFFECTING 2 LINES
----------------------------------------------------------------------
 45 | WARNING | Possible XSS detected with dvwaExternalLinkUrlGet on
    echo
 61 | WARNING | Possible XSS detected with dvwaExternalLinkUrlGet on
    echo
-------------------------------------------------------------------------



FILE: /var/www/html/vulnerabilities/exec/index.php
----------------------------------------------------------------------
FOUND 1 ERROR AND 8 WARNINGS AFFECTING 7 LINES
----------------------------------------------------------------------
  4 | WARNING | Possible RFI detected with DVWA_WEB_PAGE_TO_ROOT on
    require_once
 17 | WARNING | User input detetected with $_COOKIE.
 32 | ERROR   | No file extension has been found in a include/require
    function. This implies that some
    |         | PHP code is not scanned by PHPCS.
 32 | WARNING | Possible RFI detected with DVWA_WEB_PAGE_TO_ROOT on
    require_once
 32 | WARNING | Possible RFI detected with "vulnerabilities/exec/source
    /{$vulnerabilityFile}" on
    |         | require_once
 58 | WARNING | HTML construction with dvwaExternalLinkUrlGet detected.
 59 | WARNING | HTML construction with dvwaExternalLinkUrlGet detected.
 60 | WARNING | HTML construction with dvwaExternalLinkUrlGet detected.
 61 | WARNING | HTML construction with dvwaExternalLinkUrlGet detected.
----------------------------------------------------------------------


FILE: /var/www/html/vulnerabilities/exec/source/high.php
----------------------------------------------------------------------
FOUND 0 ERRORS AND 4 WARNINGS AFFECTING 4 LINES
----------------------------------------------------------------------
  3 | WARNING | User input detetected with $_POST.
  5 | WARNING | User input detetected with $_REQUEST.
 26 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
 30 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
----------------------------------------------------------------------


FILE: /var/www/html/vulnerabilities/exec/source/low.php
----------------------------------------------------------------------
```

```

FOUND 0 ERRORS AND 4 WARNINGS AFFECTING 4 LINES
----------------------------------------------------------------------
  3 | WARNING | User input detetected with $_POST.
  5 | WARNING | User input detetected with $_REQUEST.
 10 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
 14 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
----------------------------------------------------------------------


FILE: /var/www/html/vulnerabilities/exec/source/impossible.php
----------------------------------------------------------------------
FOUND 0 ERRORS AND 5 WARNINGS AFFECTING 5 LINES
----------------------------------------------------------------------
  3 | WARNING | User input detetected with $_POST.
  5 | WARNING | User input detetected with $_REQUEST.
  8 | WARNING | User input detetected with $_REQUEST.
 22 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
 26 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
----------------------------------------------------------------------


FILE: /var/www/html/vulnerabilities/exec/source/medium.php
----------------------------------------------------------------------
FOUND 0 ERRORS AND 4 WARNINGS AFFECTING 4 LINES
----------------------------------------------------------------------
  3 | WARNING | User input detetected with $_POST.
  5 | WARNING | User input detetected with $_REQUEST.
 19 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
 23 | WARNING | System program execution function shell_exec() detected
     with dynamic parameter
----------------------------------------------------------------------

Time: 57ms; Memory: 4MB
```