Masarykova univerzita
Fakulta informatiky

Automated SQL injection attacks validation system

Bachelor thesis

Patrik Hudák

Brno, spring 2016

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Patrik Hudák

Advisor: Mgr. Vít Bukač, Ph.D.

Acknowledgement

I would like to express my appreciation to my advisor Mgr. Vít Bukač, Ph.D. and my consultant RNDr. Václav Lorenc for their support, inputs and wisdom which helped me to write this thesis.

Abstract

The primary focus of this thesis is to study SQL injection techniques and methods of its defense. The thesis provides a design and implementation details of the system which aims to automate SQL injection attack analysis. Snort and Moloch provide network data, which is then used for the attack analysis using custom modules. The implemented system provides a web interface to present results of the analyses. A prototype of this system has been successfully deployed and tested using a tool called *sqlmap*.

Keywords

SQL injection, web application security, automation, detection, security analysis, deep packet inspection, IDS, WAF

Contents

1 Introduction

Nowadays, financial gain is a main motivation in cybercrime [Man15]. Attackers are looking to steal passwords, credit card information, personal information or intellectual property. Stolen information can be easily monetized in dark markets [Abl+14]. A major data breach can have a significant impact on a company's reputation and economic progress. When an attacker gains access to an organization, discovered information can be used to disrupt product announcements, business transactions, and stock prices.

SQL injection (described in *section 2.3*) is a common attack used to extract the content of databases without given permission. In 2015, *Trustwave* stated that 98% of web application they scanned had at least one vulnerability and 17% of found vulnerabilities were SQL injection flaws [Tru15]. *Computer Incident Response Team* (CIRT) of an organization is striving to detect and block such attacks. Network security systems are permanently monitoring the network. CIRT typically receives an alert for a possible SQL injection but usually has no further evidence to confirm whether the attack was successful or not. The evidence is missing due to the high volatility of the network data. Unfortunately, this fact presents a significant issue in some investigations.

In this thesis, I designed and implemented a system for investigating the SQL injection attacks. The system correlates data from various network sources to provide significantly more details (compared to freely available security systems) about attacks. The goal was to automate a large set of tasks that are usually done manually in SQL injection analysis. The prototype of this system has been successfully deployed and tested.

The remainder of this thesis is structured as follows. In *chapter 2*, I discuss fundamental SQL injection principles and available mechanisms for its defense. Design and analysis techniques of the implemented system are presented in *chapter 3*. The prototype implementation is discussed in *chapter 4*. In *chapter 5*, I evaluate the results of the prototype. Finally, *chapter 6* concludes and provides future plans.

2 SQL injection overview

This chapter introduces the SQL injection and available mechanisms for its defense. It also serves as a motivation for implementation part of this thesis.

2.1 Web application architecture

To understand the examples in this thesis, let's consider simple a web application:

- Application has a server-client architecture.
- Web browser acts as a thin-client.
- Web server communicates with the database server to store and retrieve data used in an application. Web application then processes the data and returns results back to the client.
- Database server can reside on the same machine (virtual or physical) as the web server or run on a separate host.

Figure 2.1 illustrates the most common architectures for modern web applications [Nel13]. The examples in this thesis will consider this architecture as the default.

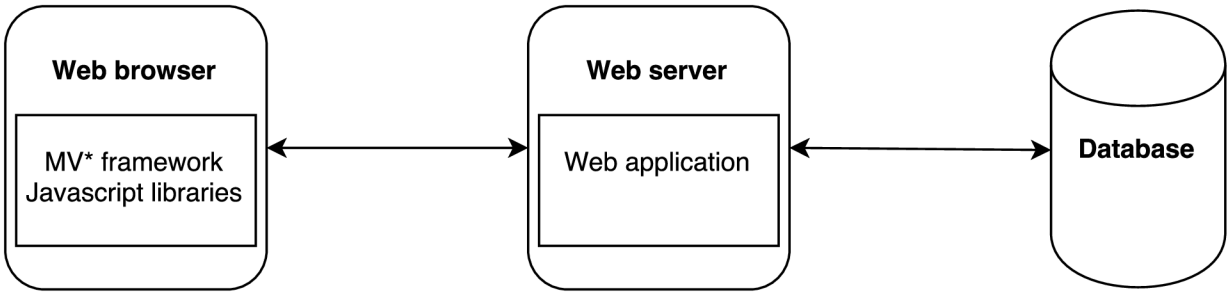

Figure 2.1: Web application architecture

2.2 SQL

SQL (Structured Query Language) is a programming language that is used to manage relational database management system (RDBMS). It is a standard language for accessing various database systems. When a web application needs to manipulate a database, it creates an SQL query for that action. The query is interpreted by the SQL interpreter and results are returned to the web application. Most common SQL operators are:

- **SELECT** – retrieving stored entries (or *rows*) from tables

```
SELECT * FROM products WHERE price < 16;
```

- **INSERT** – creating new entries in tables

```
INSERT INTO products VALUES ('T-Shirt', 14.99);
```

- **DELETE** – deleting stored entries from tables

```
DELETE FROM products WHERE price < 16;
```

2.3 SQL injection

SQL injection is one of the many attacks used on web applications. SQL injection was first documented by Jeff Forristal in 1998 [For98]. In this attack, a malicious user (attacker) manipulates an underlying database system (examples of SQL injection are available in *section 2.5*). The manipulation is done through the web application in a way that was not intended by the web application developer. SQL injection is one instance of code injection flaws. They are very common amongst all applications. When an application is vulnerable to SQL injection attacks, the malicious string that was sent the by attacker is unwillingly executed by the SQL interpreter [Cla12].

SQL injection has held its position in the OWASP top 10 risks list for several consecutive years [OWA13]. It is considered one of the well-known vulnerabilities that can be found in web applications. Nevertheless, many application programmers still forget to handle user input correctly. Basic programming rule says: **Don't trust user input**.

2.4 SQL injection risks

Users routinely provide sensitive data to web sites, like e-commerce, portals or mailbox providers. The web applications are often storing passwords in clear text. Similarly, credit card numbers along with their expiration dates and CVV codes are stored in databases. This data can often be accessed using SQL injection. Ablon et al. [Abl+14] present the findings about selling the stolen data on dark markets. For example, credit card information is being sold for prices ranging between $2 and $45.

Main risks of SQL injection are:

- **Loss of data confidentiality** – data is leaked
- **Loss of data integrity** – data is changed
- **Loss of data** – data is deleted

Database systems are sometimes running under an account with full operating system privileges (e.g. *root* for Linux). Database systems can execute operating system commands, so compromising the underlying web server can be achieved using SQL injection [Gui09]. SQL injection is often used to establish a beachhead for exploiting other, more critical servers (e.g. financial servers) in the network. Hackers are often running bots, which scan the websites for vulnerabilities. These bots can automatically compromise the web server and connect it to the hacker's network of compromised machines. Many websites run content management systems (CMS) like *WordPress* or *Joomla!*. There were many attacks on vulnerable CMS plugins and system components [Goo15]. Compromised servers are mostly used for DDoS attacks and a malware delivery until they are blocked either by the Internet Service Provider or blacklisted by security companies. Hackers are rapidly changing their malicious network using techniques like *Fast Flux*. Holz et al. [Hol+08] provide more details about such operations.

The following list provides real-world examples of successful SQL injection attacks:

- In 2013, a group of hackers known as *TeamBerserk* stole more than $100,000 from *Sebastian* customers – an Internet Service Provider based in California. Their website was vulnerable to SQL injection. Attackers were able to obtain customer details from the database and use them to collect money from credit cards [Kum13]
- Almost 6.5 million *LinkedIn* password hashes were leaked to the public. In 2012, *LinkedIn* was a victim of SQL injection attack and they spent nearly $1 million in investigation [Fon12].

2.5 SQL injection techniques

SQL injection is possible because the user input is injected into an actual SQL query without the proper validation and sanitization. Any user inputs that the application processes can be considered as potentially malicious. The malicious string can be provided through URL parameters, HTTP body or cookies (HTTP request headers in general) [Cla12]. When working with SQL injection examples, *injection context* represents the place in the SQL query where the user input is inserted (injected) [Wil16].

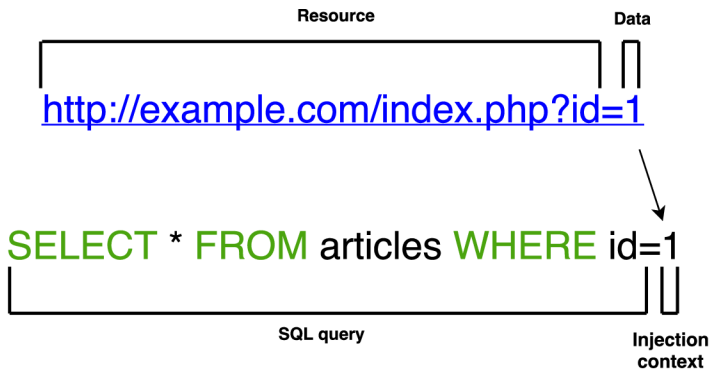

Figure 2.2: Injection context in SQL query

Several techniques are used to exploit SQL injection vulnerabilities. All examples in this section provide malicious user input as URL parameters.

- **Boolean-based blind injection**
  This method consists of carrying out a series of boolean (logical) queries against the application. An attacker observes the answers and finally deduces the meaning of such answers. She can send a malicious string, which forces the application to return a different result depending on whether the logical expression evaluates true or false. Another approach is using tautologies, i.e. logical expressions that always evaluate true to eliminate the *WHERE* operator [SP11]. Let's consider an example URL:

```
http://example.com/index.php?id=1 AND substring(@@version, 1, 1)=5
```

  In this example, the logical expression is testing whether the database system version is 5. If this is true, the underlying SQL query should return exactly one row; resource with an ID of 1. The website will be rendered without any errors. However, when the statement is false, the SQL query returns 0 results, because the logical expression evaluates as false. This can lead to standard *404 HTTP error*, or some other error message is returned. An attacker can map a database step-by-step by using such logical expressions.

- **Time-based blind injection**
  The time-based injection is often used to achieve tests when there is no other way to retrieve information from the database. This technique injects a malicious string which contains a special function or heavy query that generates a time delay. Depending on the time it takes to get the server response, it is possible to deduct some information. When the time delay is integrated into a logical expression, the attacker will be able to retrieve information from the database.

```
http://example.com/index.php?id=11;
IF SYSTEM_USER='admin' WAIT FOR DELAY '00:00:10'
```

  An example above contains a simple SQL logical expression. If the statement evaluates as true, the response will take at least 10 seconds to complete. An attacker can measure the time between request and response and thus deduct whether the expression evaluated as true (long delay) or false (short delay). This technique is very similar to boolean-based blind injection, but it can be used in situations, where the visual representation of such logical tests cannot be provided.

- **UNION query-based injection**
  The *UNION* operator in SQL is used to merge rows from two tables. Using this technique, an attacker can obtain data from various tables in a database. Suppose a simple URL:

```
http://example.com/index.php?id=1
```

Considering that the web site is a blog, this URL shows the blog post with the *id* of 1. The SQL query should return exactly one row with data like title, author, date and text. The underlying SQL query for this URL can look like:

```
SELECT title, author, timestamp, content FROM articles WHERE id=1
```

When the web application is vulnerable to SQL injection, the UNION operator can be added to the *id* parameter. The only restriction in UNION query-based SQL injection is SQL query semantics. When merging two tables together, they need to have the same number of columns and the same underlying data types. However, the number of columns can be concluded using the *ORDER BY* operator:

```
http://example.com/index.php?id=1 ORDER BY 3 -> OK
http://example.com/index.php?id=1 ORDER BY 4 -> OK
http://example.com/index.php?id=1 ORDER BY 5 -> ERROR
```

From the testing above, the table has exactly four columns. Now, an attacker can build a URL with a UNION operator:

```
http://example.com/index.php?id=1 UNION ALL SELECT user(),'0','0','0'
```

In this example, the first column is concatenated with the current database user. If the first column is a title of the article, the title will contain the current user, which can be easily recognized in the returned web page. The remaining columns are merged with string *0* to present a valid SQL query.

- **Error-based injection**
  In error-based injection, an attacker is injecting strings to generate errors due to incorrect SQL syntax or semantics. She might gather information about an application from misconfigured web frameworks that generate error messages. Some web frameworks (e.g. *ASP.NET framework*) even contain complete stack traces of affected web applications. It is then possible to enumerate table, columns, and values using these error messages.

  Example URL:

  ```
  http://example.com/index.php?id=1 or x=1
  ```

  Figure 2.3 shows the error message from the previous example. If the column x does not exist, the error message with details is generated.

Before building an SQL query to extract information, the attacker must know what data she wants to extract and where it is stored in the database. *information_schema* is a special read-only virtual database that contains metadata. Metadata include names of databases, tables, views, and their data types. Access control to information_schema varies across database systems. Every database system user can access this information, but some actions might be denied due to low privileges.

Figure 2.3: Error message from ASP.NET framework [Hun13].

2.6 Exploitation process

To exploit the vulnerability using SQL injection, an attacker or ethical hacker (performing penetration test) must carefully plan her steps. The exploitation process usually takes at least three steps:

1. **Finding the possible entry point**
   Any user-controlled parameter that gets processed by the application might be hiding a vulnerability. The entry point is usually determined by browsing the target website and finding all locations where a user can provide input. The most common locations are HTML forms. Modern applications frequently include application programming interface (API), which should be also examined.

2. **Verify the injection**
   After finding the list of possible entry points, the attacker tries to insert strings, which can result in some unexpected changes on the website (e.g. error messages). If this is the case, the examined location might be an entry point to the database.

3. **Injection**
   In this step, an attacker is building queries to directly manipulate the underlying database. First, she usually tries to enumerate the information_schema content. After successful enumeration, SQL queries to extract data can be build. One of the techniques listed in *section 2.5* is usually used.

However, this process is very exhausting when many possible entry points are found. Instead, automated tools are regularly used for some degree of automation. Finding the possible entry point cannot be fully automated. Some tools (e.g. *Burp Spider*) can at least crawl the target website and find all the pages. An attacker might examine these websites and provide a list of possible entry points. A simple way of testing entry points is through *fuzzing*. Fuzzing is a black-box testing technique, where an automated script injects a list of possible malicious strings and examines the results [Kie+09]. More sophisticated tools like *sqlmap* can test and exploit the SQL injection automatically. These tools provide a simple interface for fingerprinting and enumerating the database content. While developed primarily for penetration testing and auditing, these tools are often used by malicious attackers [Kum13].

2.7 Code-level defense

Large web applications are almost always developed using web application frameworks. They provide functions and patterns for easier development, which includes database handling and protection against vulnerabilities like SQL injection [LE07]. The well-known web frameworks such as *Django*, *CakePHP etc.* include *object relational mapping (ORM)*. ORM is a technique that enables developers to manipulate data from a database using an object-oriented paradigm. Developers are working with objects, which represent various entities in a database context (e.g. tables, rows) [ONe08]. The object changes are then automatically translated to SQL queries and executed.

When not using a web framework, web application developers should implement proper validation and sanitization of user input. The following list provides a techniques for user input handling.

- **Input blacklisting**
  This technique uses a list of potentially dangerous user inputs. If groupings of characters from the list are found in user input, a database query will not execute. This might seem to be a good method of prevention, but it is usually easy to evade. Let's consider a simple blacklisting list that contains keywords such as SELECT, DELETE, and INSERT INTO. Most database systems accept case insensitive SQL queries. An attacker can input these keywords in lowercase to successfully evade the blacklisting.

- **Input whitelisting**
  Whitelisting works in an opposite way. Only an explicitly allowed structure of the input is allowed for processing. Regular expressions are usually used for this validation. While whitelisting is significantly better approach than blacklisting, it is still not sufficient for full protection against SQL injection.

Although the previously mentioned techniques are probably the most intuitive, they are not safe. Even validated data is not necessarily safe to insert into SQL queries via string building. Preferably, the following technique should be used.

- **Parameterized queries**
  Parameterized queries enable the developer first to define all the SQL code, and then pass the value of each parameter to the query later. These queries ensure that an attacker will not be able to change the intent of a query, even if an attacker inserts SQL commands [SP11]. Parametrized queries are available in most modern libraries used for database handling.

```
// using oursql package which support qmark notation

id = get_query_param('id')
cursor.execute(
    'SELECT * FROM articles WHERE id = ?', (id, )
)
```

The previous example shows the parameterized query in Python. If the variable *id* contains parts of an SQL query, the parameterized query would not be vulnerable. The provided string would be interpreted as a value.

Administrators should follow a principle of least privilege. This principle states: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job." [Sch04] When the web application is using read operations only, write operations should be disabled. Many database systems include functions that can be leveraged by an attacker to gain additional privileges (e.g. *xp_cmdshell*). Security patches should be applied in a timely way to fix known vulnerabilities [SP11]. The unpatched software only extends the possible attack surface.

2.8 Network-level defense

In large networks, several tens or hundreds of web applications are deployed. Security analysts are looking for ways of reliably detecting web attacks. Legacy applications vulnerable to SQL injection attacks are usually present in these networks. Sometimes flaws are detected, even in the popular web frameworks. Application developers are not able to patch all applications using such framework in a short period of time. Security analysts have to detect and respond to such incidents using network-level defense mechanisms.

Despite the enormous effort of keeping endpoints and applications updated, there still might be a time frame where systems are not patched and are vulnerable to publicly available exploits. Endpoint protection software like antivirus usually cannot reliably determine whether the attack (and more specifically, web attack) occurred or not. Network security devices monitor inbound and outbound network connections to look for malicious activity, which can lead to successful exploitation. Malicious activity includes SQL injection, *denial of service (DoS)* or network scanning.

2.8.1 Firewall

A firewall is a cornerstone of almost every network. It is a network system (application or device) that filters network traffic using predefined rules [SH09]. Rules include values, which the packet must match to be forwarded in or out of the network. Firewalls usually only inspect packet header information to provide greater processing performance. Header information includes source and destination IP addresses, ports, and protocols.

There are several types of firewalls:

- **Packet-filtering firewall**
  A basic type of firewall that looks for header information on transport (L4) and network (L3) layers. Every rule is associated with an action – whether to accept or reject the matched packet when it meets the rule criteria. Packets that do not match any rule are forwarded or discarded based on configured firewall policy [SH09]. *Stateless firewall* treats each packet separately, without any previous context. While they do not require much processing resources, they can be easily evaded by techniques

like IP fragmentation [GC04]. *Stateful firewall* is tracking a state of every network connection [GL05]. Rules for stateful firewall can be extended to match a desired connection state (e.g. *ESTABLISHED*).

- **Proxy firewall**
  Proxy firewalls filter the network traffic by packet header fields on higher layers (e.g. application layer). These firewalls require much more resources than standard packet-filtering firewalls to operate efficiently and correctly. They handle network requests on behalf of internal clients, i.e. acting as a middleman to provide better security for the internal network [FS11]. Clients are usually required to use the proxy firewall as a primary gateway for the external networks. Each client on the internal network must be able to discover at least one proxy firewall. Discovery can be done by *Web Proxy Auto-Discovery Protocol (WPAD)*, which can find the best proxy firewall for a client. The most common form of proxy firewalls are HTTP proxy firewalls. HTTP proxy is routes and filters only HTTP/HTTPS traffic. Rules contain values from HTTP headers like method, user-agent or host. They can also act as a web cache, i.e. serving static content for internal hosts to provide better latency and save bandwidth. Some advanced HTTP proxies assign destination hosts and URLs a category to enable web content filtering (e.g. employees cannot access social networking sites during business hours) [BCS15].

Previously mentioned firewalls only inspect the header portion of the packets. They are not suitable for detecting web attacks like SQL injection. These firewalls can be configured to block network traffic on port 80 (HTTP), not allowing any HTTP connections. However, the whole web application is then inaccessible. Proxy firewalls can block unusual HTTP user-agents, but automated tools can easily forge this.

- **Web application firewall (WAF)**
  Unlike previously mentioned types of firewall, WAF performs a *deep packet inspection* (inspecting body part of the packets) [SH09]. WAFs protect web servers from threats like SQL injection, cross-site scripting (XSS), and many others. They can be configured to work in detection (generating alerts for rule match) or prevention (altering packets that triggers a match) mode. Some WAFs can profile the web application upfront and react to anomalies like an unusually large size of HTTP response. Others can only work as rule-matching engines. Commercial WAFs (e.g. *CloudFlare's WAF*) are often being deployed in the cloud [Reg15]. On the other side, open-source solutions are being deployed extensively. They often lack features that commercial WAFs provide (e.g. threat intelligence), but with an extensive rule set, open-source WAFs can provide a solid protection for web servers and their underlying applications. *ModSecurity* is a popular open-source WAF for Apache [Ris10].

- **Next-generation firewall (NGFW)**
  NGFW is a type of firewall that integrates many network security systems together. It includes features like deep packet inspection, SSL decryption, WAF, VPN support, and intelligence sharing [Mil11]. However, a list of features varies among security vendors. One of the leading providers of NGFW is *Palo Alto Networks* [PAN15].

2.8.2 Intrusion Detection System

Intrusion Detection System (IDS) is a system that automates the intrusion detection by analyzing network packets and looking for known (or in some cases unknown) network threats [SM07]. The main distinction between firewall and IDS is that IDS looks for both header and payload of packets (providing a deep packet inspection). Due to technological shifts and new goals in threat detection, the line between firewall and IDS is very thin. As described in *subsection 2.8.1*, WAF and NGFW perform a deep packet inspection while still having *"firewall"* in their names. An IDS consists of several devices called sensors. Sensors monitor a network and compare the traffic to a list of malicious patterns. Items in this list are referred to as *IDS rules*. When an IDS matches any of the rules, it triggers an alert which is usually handled by security analysts or network engineers. Unlike a firewall, an IDS is not actively blocking the connection by default (see *subsection 2.8.3*).

An IDS cannot detect web application attacks as accurately as WAF. WAF rule engine is optimized to handle only HTTP traffic, while IDS must support a broader range of network protocols (including HTTP). WAFs are often developed as modules to the existing web servers (e.g. *Apache* or *nginx*). Therefore, a WAF directly receives a raw HTTP packet, while IDS often needs to handle things like:

- **TCP sessions** – HTTP uses TCP on transport (L4) layer. TCP segments need to be reassembled on the destination host [PN98].
- **IP fragmentation** – The Internet Protocol can split packets into smaller chunks (called fragments) to traverse the networks with a smaller *MTU*. IP fragments need to be reconstructed on the destination host.
- **compression** – Web servers use lossless compression (e.g. *gzip*) to lower the network bandwidth.
- **TLS/SSL** – When using HTTPS, the content of the application (L7) layer is fully encrypted.

All of the above items could lead to evasion of malicious activity on IDS.

There is an extensive research on anomaly-based intrusion detection systems. Anomaly-based IDS must be trained upfront to recognize anomalies in network traffic. Training is usually done by monitoring network traffic which is proven to be *benign*, i.e. not including the malicious activity. The main advantage over a rule-based IDS (also called a signature-based IDS) is the ability to detect unknown malicious events.

To evaluate a correctness of IDS classification, four classes are defined [EES10]:

1. *True positive (TP)* – malicious event classified as malicious
2. *True negative (TN)* – benign event classified as benign
3. *False negative (FN)* – malicious event classified as benign
4. *False positive (FP)* – benign event classified as malicious

The accuracy of an IDS classification is defined as:

$$accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{2.1}$$

Numerous papers on anomaly-based detection were published [SP10; Gar+09], but converting these ideas into real-world systems is very challenging. Anomaly-based IDS usually has a high rate of false positives. Network engineers must optimize them extensively to adapt them correctly to the network environment. Signature-based IDSs are still used prevalently for a network monitoring.

2.8.3 Deployment of network devices

An IDS can be configured to operate in two modes [SM07]:

- **Detection mode** – IDS passively monitors traffic. When a malicious activity is detected, IDS will generate an alert. The traffic is not being blocked or modified in any way.
- **Prevention (or reaction) mode** – IDS can actively prevent intrusion by blocking or modifying the packet that triggered the alert. An IDS that operates in prevention mode, is called an *Intrusion Prevention System (IPS)*.

IDS / IPS can be deployed in several ways [Pap08]:

- **Inline deployment**
  Devices are deployed in a way that all inbound and outbound traffic flows through them. They are usually placed behind firewalls as an additional layer of defense. IPS can only operate in this type of deployment. Deploying a new device inline can be a very challenging task in the functional network. The inline device must be able to handle an enormous volume of traffic flowing through it and still perform an intrusion analysis. Also, the failure of an inline device can defunct the whole network. *Figure 2.4* illustrates an inline deployment scheme.
- **Out-of-band deployment**
  This is the opposite of an inline deployment. The device can be placed on a regular endpoint and network traffic that needs to be checked for malicious activities is forwarded to it. Forwarding can be achieved by configuring switches to replicate all packets to one particular port. This is called *port mirroring*. A *network TAP* can also be used for passive monitoring [Wir]. An IDS is connected to mirrored port (or TAP) and thus receives and analyzes the traffic passively. An IPS cannot work in an out-of-band deployment scheme since it needs to alter the network traffic actively. *Figure 2.5* illustrates an out-of-band deployment scheme with port mirroring.

Firewalls (described in *subsection 2.8.1*) are almost always placed inline. Similarly, WAFs are supposed to work in a prevention mode (deployed inline) to actively block web attacks. IDS usually requires much greater processing time for detection than a firewall

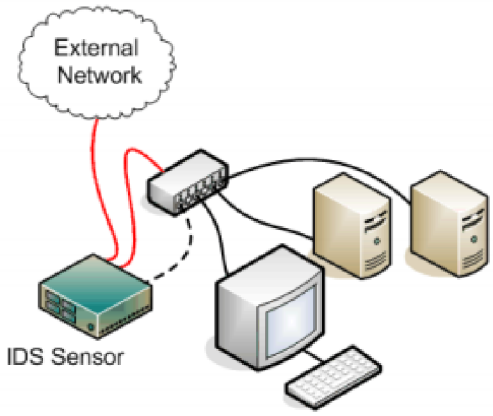

Figure 2.4: Inline deployment [Pap08]

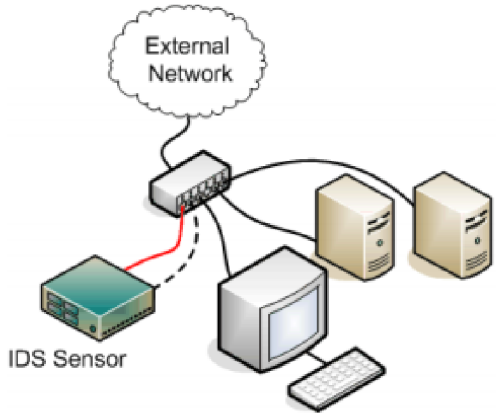

Figure 2.5: Out-of-band deployment [Pap08]

or WAF. WAFs are only responsible for HTTP traffic whereas IDS are dealing with basically every network protocol. Hence, IPS needs to process all the inbound and outbound traffic (usually several gigabits/s), or it will be the primary performance bottleneck for a monitored network.

Code-level defense is the best way of preventing SQL injection. However, network-level defenses are still needed to provide an additional layer of security. From the presented network-level systems, a WAF is best choice for the web application security. Some network environments (e.g. large corporate networks) host many application servers without a maintained inventory of deployed web applications. Deploying a WAF would be almost impossible because it needs to be placed in front of every web server. WAFs can also work in the cloud – DNS entries must be updated to point to the cloud's WAF provider. Unfortunately, most intranet web servers are non-routable from the Internet; therefore, cloud-based WAF providers cannot be used. An attacker might use SQL injection to gain access to the publicly available servers of an organization. She can use them as pivots for accessing the organization's private network, where the valuable data reside.

3 System design and Analysis

Although network security devices can detect SQL injection, they provide very little or no evidence for its investigation. The primary goal of investigating these attacks is to determine, whether the SQL injection was successful, i.e. whether the data was leaked from the database or not. Indeed, another network mechanism can collect all sort of evidence for these cases. Security analysts usually need to correlate events from various system to get better visibility into an attack.

In the implementation part of this thesis, I created a system for automating this correlation. This chapter presents the decisions that I made to design such system.

3.1 Architecture

3.1.1 Requirements

The following list contains the functional requirements for the system:

- Existing open-source network security devices should be leveraged if possible.
- All components must be deployed out-of-band (more details in *subsection 2.8.3*).
- The architecture should be modular.
- The analysis results should be provided by a web interface.

3.1.2 High-level architecture

This section presents the high-level architecture of the implemented system, based on the requirements for the system. The implemented system is designed in a modular architecture which allows other developers to enhance its functionality. The implemented system is composed of several subsystems (described briefly in the list below). The following sections of this chapter present every subsystem in greater detail.

- **Detection system**
  The detection of SQL injection attacks (*section 3.2*) is handled by an IDS deployed out-of-band. It contains rules to detect and alert on suspicious HTTP requests or responses. More specifically, I chose to use Snort (*subsection 3.2.1*) as a default detection mechanism. Snort is still widely deployed, and a large set of rules is freely available. Although Snort was chosen to be a default detection mechanism for the implemented system, it can be easily replaced by alternative detection mechanisms (*subsection 3.2.2*).
- **Traffic acquisition system**
  As will be discussed in *section 3.3*, a system for full packet acquisition is required. Acquiring the network traffic is used to gain better visibility into an SQL injection attacks. *Moloch* (subsection 3.3.1) was chosen as a default system for the packet acquisition.

- **Analysis engine**
  The analysis engine (*section 3.4*) is used to analyze the suspected SQL injection attack. It retrieves a packet capture (PCAP) from Moloch and performs a various test on it.
- **Database**
  A *document-oriented NoSQL database* (*section 3.5*) is used to store analysis results.
- **Application Programming Interface**
  Application Programming Interface (*section 3.5*) is used to provide a connection between above mentioned subsystems. It is also used for accessing the analysis results that can be processed by external systems.
- **Web interface**
  A web interface (*section 3.5*) is used to present the analysis results.

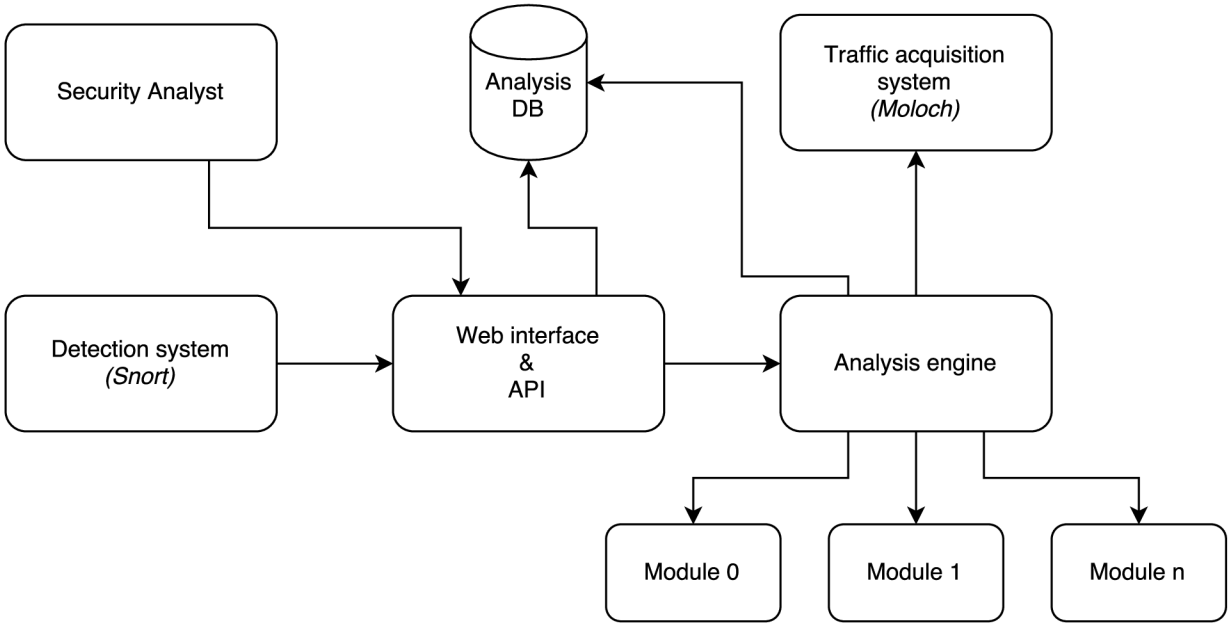

Figure 3.1: High-level system architecture

From the above list, *analysis engine*, *modules*, *API*, and *web interface* were created in the implementation part of this thesis.

3.2 Detection

The first step in analyzing the SQL injection is determining what exactly should be analyzed. Although the system could analyze every HTTP packet in network traffic, that would be excessive due to the huge traffic volume in some networks. The analysis engine performs time-consuming tasks that cannot be executed in real-time. An IDS is used for analysis prioritization. Unlike WAF (*subsection 2.8.1*), IDS can be deployed out-of-band (in

detection mode). While it is feasible to use any IDS for detection because of the modular architecture, I chose to use Snort as a default option.

3.2.1 Snort

Snort is an open-source network IDS created by Martin Roesch [Roe99]. *Figure 3.2* illustrates the high-level architecture of Snort. Preprocessors process packets before they reach the detection engine. They usually normalize packets for easier rule matching. Preprocessors can detect protocol anomalies and trigger the alert even before the packet is forwarded to the detection engine. For instance, *stream5* preprocessor handles TCP stream reassembly, and *frag3* is used for IP defragmentation.

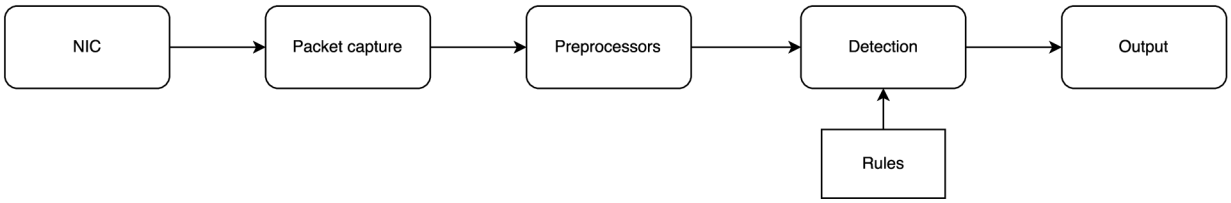

Figure 3.2: Snort architecture

Snort rule is a set of keywords used for identification of known network attacks. The detection engine loads provided rules and compares it to every packet it receives from preprocessors.

Example of Snort rule which alerts on every inbound *ECHO REQUEST* ICMP packet:

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP PING"; icode:0;
itype:8; classtype:misc-activity; sid:384; rev:5;)
```

Every Snort rule consists of [DH12]:

- **Header**
  Rule header provides basic data for matching a rule against packets.

Action	Protocol	Src IP	Src Port	Direction	Dst IP	Dst Port

Table 3.1: Snort rule header

  Some fields can contain variables (e.g. *$EXTERNAL_NET*) instead of fixed values. Variables make rule management much easier. The string *"any"* can be used to match every possible value.

- **Body**
  The rule body is enclosed in the parentheses. It specifies advanced conditions like payload or HTTP headers. A body is composed of several *options*. An option is structured as a *keyword* and *argument*, separated by a colon. Options are separated by a semicolon. Rule body also includes description message (option *msg*), Snort

ID (option *sid*) and rule revision number (option *rev*). Snort allows some keywords to be specified as regular expression (PCRE). However, PCRE does not perform as fast as standard content matching does.

When the Snort matches network traffic against any rule, an alert is generated. Generated alert (using *alert_fast* output module) for the rule in the example above:

```
[**] [1:384:5] ICMP PING [**]
[Classification: Misc activity] [Priority: 3]
04/13-03:12:08.359790 10.0.1.10 -> 10.0.1.254
ICMP TTL:64 TOS:0x0 ID:38125 IpLen:20 DgmLen:84
Type :8 Code :0 ID :32335 Seq :1 ECHO
```

The first line provides information about matched packets. It starts and ends with [**] character sequence. The three values displayed inside the brackets (separated by a colon) are:

1. **Generator ID (GID)** – specifies, which module produced the alert. Detection engine has GID 1, and each preprocessor has its own GID.
2. **Snort ID (SID)** – unique identification of the rule that triggered an alert.
3. **Revision number (REV)** – revision number of the rule that triggered an alert.

The remaining lines provide general information about the host and packet that triggered the alert. Along with alerts, Snort is by default configured to log the full content of packets that triggered the alert. Packets are usually logged to separate files.

3.2.2 Alternatives

Several other detection mechanisms exist. The following list provides brief details about alternatives that might be used for the detection.

- **Suricata**
  Suricata is an open-source network IDS, owned by the Open Information Security Foundation (OISF). It is very similar to Snort. Snort rules are compatible with Suritaca detection engine. Suricata allows the rules to be extended with application layer (L7) protocols [Lan16], mainly HTTP, FTP, and DNS. Protocol-specific keywords (such as *http_method* for HTTP) can then be used directly in the rule body for better clarity.

  Example of Suricata rule:

```
alert http $EXTERNAL_NET any -> $HOME_NET any
(msg:"ET WEB_SERVER SQL Injection Attempt (Agent CZxt2s)";
flow:to_server,established; content:"User-Agent|3a| czxt2s|0d 0a|";
nocase; http_header; reference:url,doc.emergingthreats.net/2011174;
classtype:web-application-attack; sid:2011174; rev:4;)
```

- **Bro**
  Bro is a platform for network traffic analysis with a focus on security. It is often compared to traditional IDSs like Snort or Suricata. Unlike previously mentioned systems, Bro does not support rules detection by default. Bro provides a Turing-complete language for scripting purposes [Bro]. This approach makes it more event-based than a signature-based system. In the scripts, Bro allows hooking network events (e.g. when a HTTP request with GET method is observed, call method x). This way, more advanced detection mechanisms can be implemented. Bro comes with built-in support for many application-layer protocols (HTTP, DHCP, SSH *etc.*). These are usually provided as an API to be used by user-defined scripts.

  The following example contains a basic Bro hook. The function is called every time a HTTP request is observed. The request details are provided as function arguments.

```
event http_request(c: connection, method: string, URI: string,
                   unesc_URI: string, ver: string) {
    // arguments processing
}
```

- **Haka**
  Very similar to Bro, Haka defines its own scripting language (based on *Lua*) to describe protocols and policies (scripts written in Haka language) [Hak]. The protocol dissector and policies are then applied either on live network traffic or captured PCAPs. Haka can interpret the protocol grammar and follow the states and transitions based on the underlying state machine. Policies can detect and respond to various events and conditions. Unlike Bro, Haka can be deployed in-line, i.e. traffic can be actively altered using provided policies. Packet injection is also possible. *Hakabana* is a tool built on *ElasticSearch*, which visualizes network traffic going through Haka in real-time.

```
local ipv4 = require('protocol/ipv4')

haka.rule{
    hook = ipv4.events.receive_packet,
    eval = function (self, pkt)
        if pkt.src == ipv4.addr("192.168.0.1") then
            pkt:drop()
        end
    end
}
```

Listing 3.1: A Haka policy example. Every packet with source IP address of *192.168.0.1* is dropped.

Extensive documentation is available on the websites of individual projects. Due to the previously mentioned modular architecture, Snort can be replaced by any of these systems.

Another option is to use a logging server (e.g. *Splunk* or *ELK*), which aggregates the data from various security systems and notifies the analysis engine after some threshold of events is exceeded [Rob13]. The implementation details are presented in *section 4.2*.

3.3 Traffic acquisition

Although IDSs are very powerful in detecting the malicious activities, they lack the ability to provide detailed PCAP for the intrusion. To analyse a possible SQL injection, full PCAP of the network traffic during the time frame of an alert needs to be acquired. An IDS can be configured to log the packet that matched the signature. However, relying solely on this data would be insufficient, because SQL injection attacks are usually generate a large set of similar HTTP requests. An SQL injection attack might start several packets before the first IDS detection. These packets are not logged by an IDS, mainly due to the performance reasons (e.g. using a big buffer for packets). A separate mechanism for traffic acquisition must be used.

Libpcap is a standard UNIX library for capturing and filtering network packets [DH12]. Many popular tools such as *Wireshark*, *nmap* and *tcpdump* are based on libpcap. It also includes very powerful packet filtering language called *Berkley Packet Filter (BPF)*.

Example of BPF query:

```
tcp and port http and dst host 192.168.1.100
```

This query filters traffic with port 80 (HTTP) and destination IP of 192.168.1.100.

One of the most widely used tools for traffic acquisition is *tcpdump*. Tcpdump can record and filter the network traffic using BPF in real-time. Although tcpdump does not have a graphical user interface, it can be used for quick traffic acquisition and analysis. Tcpdump comes with several limitations for purposes of SQL injection attack analysis:

1. It does not scale well. There is no centralized database where the acquired traffic is stored.
2. It does not understand higher-layer protocols like HTTP. Thus, tcpdump does not provide an ability to filter traffic using HTTP headers.

Because of these shortcomings, I chose to use *Moloch* for traffic acquisition.

3.3.1 Moloch

"Moloch is an open source, large scale IPv4 packet capturing (PCAP), indexing and database system." [Mol] It allows fast filtering, searching, browsing, and exporting stored PCAPs using a simple web interface or API. Moloch captures (acquires) the network traffic and groups it into sessions. However, there is not any formal definition of what a Moloch session is. From my observation, it can be defined as a 7-tuple:

$$session = (startTime, stopTime, srcIP, dstIP, srcPort, dstPort, protocol) \tag{3.1}$$

This approach allows easier grouping of similar requests. One session is seen as one connection. However, several SQL injection tools use techniques, which Moloch cannot cope with very well. For example, *sqlmap* creates a new TCP connection for a new request. Thus, a different source port number is used for every sqlmap request. In this case, the new session is created for each request. To retrieve the full PCAP for sqlmap connection, multiple sessions need to be combined. Fortunately, Moloch provides a simple yet powerful query language for building expressions (similar to BPF) to filter the acquired traffic and combine sessions together.

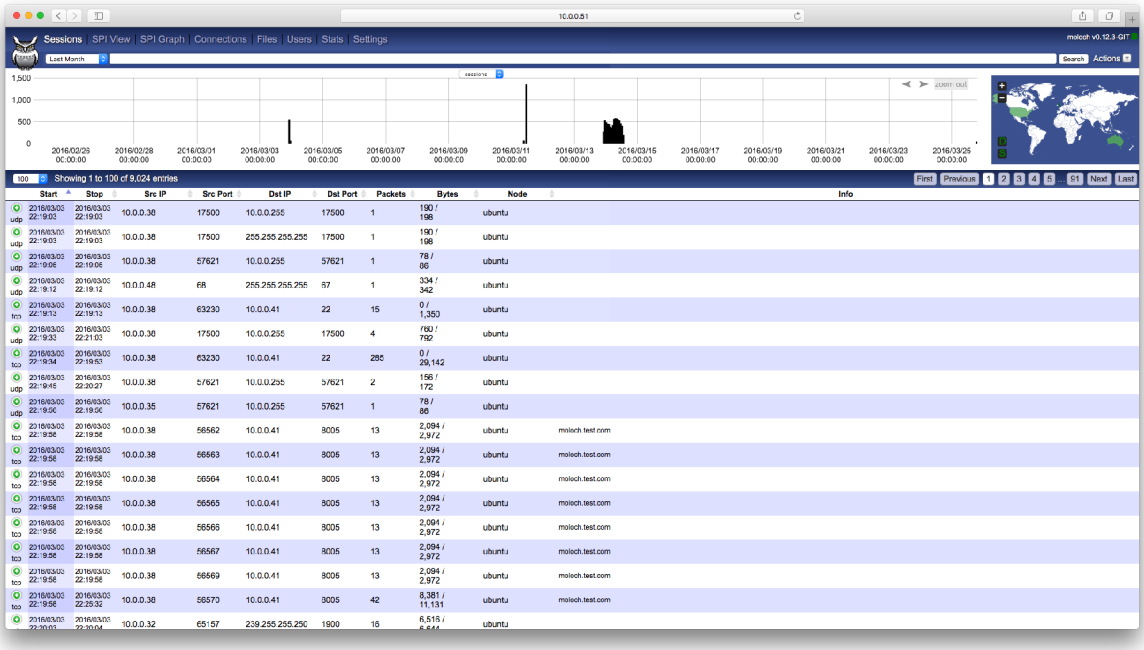

Figure 3.3: Moloch web interface

Moloch is a complex system that consists of three main parts:

1. **ElasticSearch database** – indexing stored sessions. Tcpdump only stores raw PCAP files, whereas Moloch also stores an extensive index (using *ElasticSearch*) of captured sessions. This index allows one to perform almost real-time searches among a large volume of network traffic.
2. **Capture** – a separate process that captures traffic from the network interface using libpcap.
3. **Viewer** – an interface for viewing, filtering and exporting stored sessions. It also provides a REST API.

Moloch is also able to understand HTTP traffic. If the HTTP session is acquired, Moloch automatically applies *HTTP tags* to it. Moloch tags are strings that are associated with sessions for the filtering purposes. In this way, a filtering expression can contain HTTP-specific keywords and fields.

Simple Moloch filtering expression:

```
ip.src == 10.0.0.38 && starttime >= "2016/03/03 22:19:03"
&& port.dst == 17500
```

Filtering expression using HTTP tags:

```
protocols == http && http.method == GET && http.statuscode == 200
&& stoptime <= "2016/03/04 00:00:00"
```

3.4 Analysis engine

The analysis engine integrates Snort and Moloch together and analyzes the possibly malicious traffic. It does not try to give a discrete value for the resolution, i.e. whether the network traffic is malicious or benign. Rather, the analysis engine examines the traffic to obtain information that might be interesting for the attack investigation.

After receiving the alert from Snort, the analysis engine requests the full PCAP from Moloch. The actual analysis is performed using standalone scripts called *modules*. Each module performs certain operations on the PCAP and gives the output back to the analysis engine. Because of the modular architecture, modules can be added, removed, and changed easily. A database stores outputs from the modules for the next processing (such as attack investigation). Results of the analyses can be accessed through a web interface or API, described in *section 3.5*.

The following list serves as a description of the *analysis workflow*. The list contains an explanation of the core modules included in the analysis engine. The output from these modules is a *JSON* object with no defined structure. Each module defines the structure of its output, which is then processed by a web interface.

- **Attacker IP address details**
  One of the fundamental data that the analysis engine provides is details about the (possible) attacker. An IP address registered in Russia or Ukraine does not directly imply that the SQL injection alerts are true positives, but it might be a red flag. Goncharov [Gon12] presents details about hacking operations in these countries. The same principle applies to discovering that the IP address belongs to the Tor exit node. Tor is an anonymous network that routes encrypted traffic through different nodes [DMS04]. Attackers often use Tor to hide their locations. According to *CloudFlare*: "Based on data across the CloudFlare network, 94% of requests that we see across the Tor network are per se malicious." [Pri16] The collected details about IP address are:
  - IP address owner (ISP) / Autonomous system
  - Origin country
  - CIDR notation
  - Abuse e-mail contact
  - Whether the IP address is a Tor exit node

Everything except the last item can be retrieved from the Regional Internet Registries (RIRs) using *whois* protocol. The list of Tor exit nodes is publicly available. The Tor exit node passes the encrypted traffic to the Internet. Comparing the IP address against this list will determine whether the connection was made over Tor.

- **Web server details**
  During investigation, it is sometimes useful to have basic details about the target web server. Because the analysis engine does not have direct access to the analyzed web server, it needs to collect these details using pattern matching. *Wappalyzer* [Wap] is an open-source tool able to detect underlying technologies used on a website. It usually finds information about the web server, CMS (if used), Javascript libraries and much more. The detection is possible by comparing the HTTP headers, URL structures, and web server responses against the database of known patterns. The analysis engine utilizes the Wappalyzer passively, i.e. it is not making new connections to the web server. Rather, web server data from the PCAP are extracted and analyzed by the Wappalyzer. This approach also follows basic operational security rules. The URL that is extracted from the PCAP can contain query parameters that include SQL statements. Requesting such URLs using Wappalyzer can accidentally result in SQL injection (e.g. altering with the database).

- **Statistical analysis**
  An attacker usually fingerprints the database using only one entry point (discussed in *section 2.6*). When analyzing the PCAP with SQL injection, several patterns can be observed. An attacker is usually requesting the same URL with different query parameters or HTTP body (testing various SQL commands). The analysis engine tries to recognize such patterns in the traffic. It finds the single most used URL endpoint and compares the number of requests to this endpoint against the total number of requests. A high ratio from this equation might indicate an SQL injection attempt. Next, the analysis engine tries to extract the data portion of these requests and examine their semantics. If the parts of SQL commands are found, this is good evidence that an SQL injection attempt occurred [Mey08]. However, the analysis engine is not able to determine whether the SQL injection attempt was successful or not. The analysis engine also looks for HTTP status codes in responses to such requests. When application error handling is correct, it should return *404 code* to requests containing SQL injection attempts. Unfortunately, this is not always the case. On the other hand, automated tools are usually not fully-customized browsers. When running such tool, it only requests the HTML of the website, not the external dependencies like Javascript scripts or CSS stylesheets provided in this HTML. The analysis engine tries to find such behavior by comparing the Content-type from the server responses. If the *text/html* dominates the PCAP, it is very likely that a command like HTTP client was used.

- **Database canary**
  When enumerating the database tables, an attacker usually tries to query the *information_schema* database (see *section 2.5*). Automated tools that are capable of retrieving the table names and columns (e.g. sqlmap) try to gather all sort of details from information_schema. To determine whether the content of information_schema

was retrieved (whether SQL injection was successful), the analysis engine can use database canaries. Canary is a sufficiently long string that is placed in the information_schema database. When the string is present in the server response, it is an indication that SQL injection was successful. While the information_schema is holding a database metadata, a canary can be placed by creating an empty table with such name. This process needs to be done manually by a database administrator. Because many database servers might be present in the network, multiple canaries can be generated. Canaries are linked with a description which allows to easily determine the affected database server. A generated canary is a string with 256 bits of entropy (64 characters with 2 characters per byte) which is stored in the analysis database. During the analysis, the analysis engine is scanning server responses, looking for generated canaries. This method will not work for time-based blind injection and queries using write operations. However, for attacks that involve data leakage, it is very likely that the tables will be enumerated.

- **Connections**
  While Moloch usually captures network traffic for the whole network subnet, it can be used to provide a list of other hosts that attacker has communicated with. During SQL injection attack, an attacker might try to exploit another web server located on the same subnet. Moloch is able to provide such information through its API. It generates the graph of connections where nodes are network hosts and edges represent communication between these hosts.

3.5 Storage and Web interface

The internal database used by the analysis engine and a web interface is *RethinkDB*. RethinkDB is document-oriented NoSQL database focused on real-time applications [Ret]. The NoSQL database was chosen due to a fact that the analysis engine provides an unstructured JSON for each analysis. Expressing the analysis results would be problematic in a traditional relational database. RethinkDB comes with simple query language and web interface for easier testing and debugging. RethinkDB instance can be run on a separate machine because remote access is possible using *ReQL wire protocol*.

```
cursor = r.table('analyses').filter(r.row['dst_ip'] == '10.0.0.1').run()
for document in cursor:
    print(document)
```

Listing 3.2: RethinkDB query in Python. It returns records from *analyses* table filtered by *dst_ip* key.

A web interface gives access to the analysis results. It is composed of several individual pages:

- **Dashboards**
  They provide a visualization of data from analyses.
- **List of analyses**
  All analyses are listed in a compact table. Each analysis can be expanded to provide detailed results.
- **Analysis result**
  Individual analysis results provide every detail of the analysis. Every module defines its own visual representation. For example, a module that retrieves IP address details visualizes its results in the form of a map and table. The output from these modules is separated using tabs.
- **Canaries management**
  This page is used to list stored canaries and generate new ones.

Screenshots of the web interface are available in *Appendix A*.

Application programming interface is implemented along with a web interface. It allows the external systems to access analysis results for further processing. API also represents a link between Snort and analysis engine (details are provided in *section 4.3*).

4 Implementation

This chapter details the implementation of the system by focusing on providing the fundamental information such as a configuration of subsystems and used libraries.

Python was chosen as the primary programming language for two reasons. First, it provides very readable syntax. Second, many open-source packages related to the SQL injection analysis exist in Python.

4.1 Configuring prerequisites

There are many different deployment strategies for Snort and Moloch. Installation guides for these systems can be found on their websites. Therefore, I will focus mainly on the configuration of these systems related to the implementation.

After installation, Snort provides no ability to detect complex SQL injection attacks. Freely available Snort rules from *Talos* [Tal] and *Proofpoint* (formerly *EmergingThreats*) [Pro] provide a good set of rules for detecting web attacks.

Rule updates can be performed manually or automatically. Manually updating large numbers of rules may be prone to errors or misconfigurations. *PulledPork* [Die15] is an open-source tool used for rule management automation. PulledPork can download rule updates, add them to running Snort instance, and generate signature maps.

Snort is able to produce a fast binary format of logs called the *unified2*. Unified2 can be configured to log packets, alerts, or both of them [Uni]. The main reason behind using binary output format is speed and efficiency. Other applications should convert a unified2 log to the desired format. Snort also allows the use of additional, more user-friendly output modules:

- **alert_syslog** – sends alerts to the *syslog*
- **alert_fast** – prints one file summary of the alert
- **alert_full** – prints full alert message
- **alert_unixsock** – sends alert using UNIX socket
- **log_tcpdump** – logs malicious packet into format that can be analysed using network tools (such as tcpdump or Wireshark)

Any of these output modules (including unified2) can be used with the analysis engine. Processing these alerts for the analysis engine is covered in *section 4.2*.

To correctly associate the events from Snort to the sessions in Moloch, few things need to be set-up in front:

- **Deployment point**
  Moloch needs to cover the same network subnet as Snort. When an alert is generated for the host, the analysis engine needs to retrieve the packet capture for this host. If Moloch capture is deployed on the different network subnet, the analysis will fail.

- **Timestamps**
  Both systems (Snort and Moloch) must log timestamps in a *UTC standard*. They must use common *NTP server* to provide an accurate time synchronization.
- **Moloch's REST API**
  REST API is enabled by default, but it might be blocked by a network firewall (e.g. if Moloch viewer is running on different network segment). Moreover, Moloch's viewers are using a self-signed SSL certificate. They should be replaced by certificates signed by the official certification authority, or the certificate validation must be turned off.
- **PCAP rotation**
  Capturing network traffic in large networks can generate terabytes of data every day. Configuring Moloch to delete old sessions will provide a free space for storing new sessions in the future.

4.2 Alert forwarding

To start the analysis, a Snort alert needs to be forwarded to the analysis engine. As described in *section 4.1*, Snort can produce logs in various formats. I used the *unified2* format as a default option for the logging purposes. Firstly, the unified2 logs must be converted into some structured data format. *Barnyard2* [Die15] is an open-source tool typically used for this task. Barnyard2 is capable of monitoring the Snort logs for changes and converting newly created entries into specified format. However, I decided to use an open-source Python package called *idstools* [Ish16] for converting logs. The main reason behind this decision is higher flexibility in processing unified2 logs directly using Python.

A directory where Snort logs reside needs to be constantly monitored for new entries. Monitoring the whole directory instead of individual files is required due to the log rotation activity performed by Snort. An *idstools* package includes a simple interface for continuous monitoring.

```
from idstools import unified2

reader = unified2.SpoolEventReader(directory, prefix, follow=True)
for event in reader:
    # process event
    # ...
```

Listing 4.1: Using unified2 log reader from *idstools* package

The code in *listing 4.1* will loop over the Snort logs without terminating (because of *follow=True* argument). When Snort adds a new alert to the log, reader object is notified and creates a new iteration. This activity is also called *spooling (simultaneous peripheral operation on line)*.

A variable *event* is an object containing parsed fields from the currently processed Snort alert. The most relevant fields from this object are:

- **timestamp** – Time of an alert provided as a UNIX timestamp.
- **source-ip** – Source IP address. It is interpreted as the IP address of an attacker.
- **sport-itype** – A source port number is ignored for reasons explained in *subsection 3.3.1*.
- **destination-ip** – Destination IP address. It is interpreted as the IP address of a targeted web server.
- **dport-itype** – Destination port number. It usually represents a port number of a targeted web server.
- **generator-id, signature-id, signature-revision** – GID, SID, and REV. These fields are described in *subsection 3.2.1*.

The fields from the previously provided list are extracted to a Python dictionary. However, a signature identification that triggered an alert is provided only as numeric values. Indeed, this representation is fine for a machine, but security analyst should receive a textual description of an alert. Fortunately, an *idstools* package provides a module for mapping numeric signature values to more meaningful textual descriptions. Example of this mapping is provided in *listing 4.2*. An output of the mapping is included in Python dictionary with alert fields. To use this functionality, two files must be available:

- **sid-msg.map** – Provides mapping from a numeric signature ID (*SID*) to a textual representation. *PulledPork* (presented in *section 4.1*) can automatically generate this file after updating the rules.
- **gid-msg.map** – Provides mapping from a numeric generator ID (*GID*) to a textual representation. It is usually available after installation in the Snort configuration directory.

```
from idstools import maps

sigmap = maps.SignatureMap()
sigmap.load_signature_map(open('/path/to/sid-msg.map'))
sigmap.load_generator_map(open('/path/to/gid-msg.map'))

# get textual representation of the signature
sigmap.get(gid, sid)
```

Listing 4.2: Using signature mapping from *idstools* package

The alert forwarder performs two additional tasks:

- **Signatures filtering**
  The alert forwarder should only forward the alerts with relevant signatures. The forwarder contains a list of signature IDs related to SQL injection attacks. If a signature ID of a received alert is not present in this list, an alert is immediately discarded. This list is fully customizable and can also include user-defined rules. The disadvantage is that rules need to be manually examined to create the relevant signature list.

- **Bookmarking**
  The alert forwarder also needs to keep track of events that were already forwarded to the analysis engine. When the alert forwarder is restarted (e.g. due to the system reboot), it should not forward all stored alerts, just the new ones. An *idstools* package can cope with this issue by using a simple technique called *bookmarking*. It works by saving the offset of the last read Snort alert to the separate file. This offset is updated after each reader iteration. The bookmark saves the offset of all Snort log files (in a case of log rotation). When the reader object is created, the bookmark assures, that only new entries will be processed. To enable the bookmarking, a *bookmark=True* argument is added to the reader object's constructor.

After the event is converted to Python dictionary, the analysis engine is contacted. The communication is accomplished using *HTTP*. Python dictionary with alert details is converted to JSON and added to HTTP body. The alert forwarder then makes HTTP POST request to the API. *Figure 4.1* illustrates the simplified workflow of the alert forwarder.

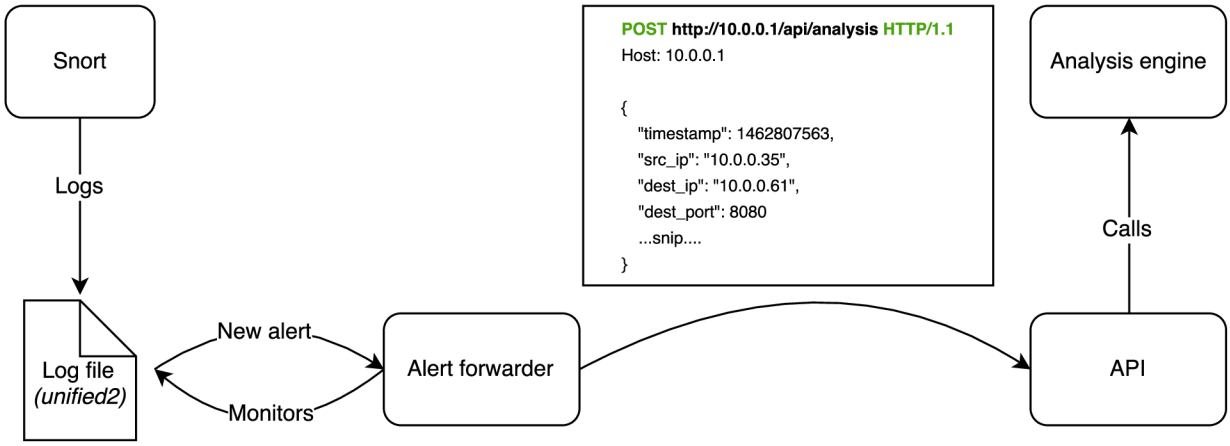

Figure 4.1: Alert forwarder workflow

It is important to note, that all principles presented in this section can be applied to any detection mechanism from *section 3.2*. To use the alternative detection mechanism, the alert forwarder for this mechanism needs to be implemented. The template for forwarder creation can be found in the attached archive (*scripts/forwarder_template.py*).

4.3 Application Programming Interface

An API is implemented in *Flask*. It uses HTTP as a communication protocol. Flask [Ron16] is a popular web framework written in Python. A developer defines own functions to handle HTTP requests on specific URLs (also called API endpoints). After Flask receives a HTTP request, it forwards the request to correct handler. The handler processes the requests and sends back a response.

4.3.1 Creating a analysis

During the SQL injection attack, an attacker usually generates more than one IDS alert. When attacker starts an SQL injection, it takes some time to test all queries either manually or using automated tool. In order to assess the whole scope of SQL injection attack, we need to analyze all queries altogether. Let's consider the scenario, where the alert is *true positive*, and analysis is started right after receiving the first alert. The analysis engine would have very little, or no data to work with because an attacker has just started the enumeration. Also, it takes Moloch some time to index the acquired traffic properly. It is possible, that even if Moloch has successfully captured the traffic to the PCAP on the filesystem, the analysis engine would not be able to retrieve the traffic using API. This case can happen in situations when there is little time between capturing the traffic and requesting it.

Therefore, the API handler for alert processing will not instruct the analysis engine to create an analysis for the same host during the time between the first alert and the estimated start of the analysis (15 minutes from the first alert by default). SQL injection alerts generated in this time frame are aggregated (grouped) under one analysis. When the alert forwarder sends a new alert, the API handler checks whether there is a planned analysis for the similar alert. This check is achieved using a cache of alerts. This cache is a Python list, where items expire (are deleted) after the analysis for the item has started. Items in the cache are simple 3-tuples:

$$alert = (srcIP, dstIP, dstPort) \tag{4.1}$$

If the 3-tuple for new alert exists in the cache, the API handler adds an alert to the existing analysis. In another case, a new analysis is created.

4.3.2 Task queue

After the API handler finishes the alert aggregation, it is ready to start the analysis engine. However, API handler cannot simply invoke an analysis engine because the current execution thread would be blocked for the time of the analysis. The API needs to be constantly available for handling the incoming HTTP requests. One solution for this problem is to create a new execution thread per analysis. However, this approach is hardly maintainable and it is difficult to scale. That is why I decided to use task queue for the analysis delegation. Task queues are used to distribute the task execution across processes and machines.

Celery [Sol+16] is a highly customizable asynchronous Python task queue. Celery allows using various message brokers (e.g. *Redis*, *RabbitMQ*) that are used to distribute tasks using message passing. Workers are processes which receive tasks, execute them, and provide the results back to Celery. Celery can equally distribute tasks to available workers. Workers can be running on different machines as long as they can connect to the message broker.

The API handler creates a new analysis by creating a new task in Celery. The execution of the task is immediately postponed for reasons explained previously. Celery selects the worker, schedules the execution but does not invoke the analysis engine. Next, the API handler creates an analysis entry in a database. The entry contains a list of aggregated alerts, the current timestamp, and task status information. In this phase, the status of the task is set as *PENDING* – the analysis is scheduled but has not started yet.

API documentation is available in *Appendix B*.

4.4 Analysis engine

When Celery decides that the scheduled task is ready, the selected worker starts executing the task. The analysis engine is started and performs the following steps:

1. **Changes the analysis status**
   The status is changed from *PENDING* to *PROGRESS*.

2. **Retrieves the PCAP from Moloch**
   The analysis engine requests PCAP containing all HTTP traffic that is exchanged between source IP and destination IP and has destination port that was observed in the initial alert. Moloch API (*/sessions.pcap* endpoint) is used for this operation. By default, Moloch uses HTTP Digest Authentication for API calls. Python's *request* package is used for HTTP processing. The analysis engine creates a Moloch expression which filters the correct traffic. The expression template follows:

   ```
   port.dst == {dst_port} && protocols == http &&
   ip.src == {src_ip} && ip.dst == {dst_ip}
   ```

   In simple words, the analysis engine requests PCAP containing all HTTP traffic that is exchanged between source IP and destination IP and has destination port that was observed in initial alert. Along with the expression, *startTime* and *stopTime* parameters are used. They define the time range for requested sessions. All the parameters (expression, startTime, stopTime) are encoded as URL parameters. The analysis engine stores the retrieved PCAP in a specified directory.

3. **Calls the modules**
   Modules are called sequentially. See *subsection 4.4.1* for more details.

4. **Updates the database entry**
   Results from the modules are collected and stored in the database. The task status is changed from *PROGRESS* to either *SUCCESS* or *ERROR*, depending on the execution of the analysis engine. The error might occur for example when Moloch

is not reachable, or the none of the modules finished successfully. The example of JSON for analysis is exposed in *Appendix B*.

4.4.1 Modules

Modules are used for actual processing of alert details and PCAP. Modules reside in the separate Python package from which they are dynamically imported. The list of used modules can be fully customized by changing *__all__* variable in *modules/__init__.py* file. Every module that is being imported must contain *Module* class with *__init__* and *bootstrap* methods. The analysis engine passes the parameters to the module as follows:

```
for module in analysis_modules:
    module_results = module(opts, pcap_path, config).bootstrap()
```

- **opts** – dictionary with alert details (source IP, destination IP *etc.*)
- **pcap_path** – location in a file system, where retrieved PCAP is stored
- **config** – dictionary with configuration details (RethinkDB host and port, Celery broker *etc.*)

The *bootstrap* method should return the Python dictionary that will be stored in the database. The structure of this dictionary is not defined.

The following list contains implementation details for modules, which were presented in *section 3.4*.

- **Attacker IP address details**
  Attacker IP address is extracted from *opts* dictionary. Then, *ipwhois* Python package is used to retrieve the whois information from RIRs. List of current Tor exit nodes is downloaded from the publicly available source.
- **Web server details**
  While Wappalyzer is not written in Python, *python-Wappalyzer* [Hor] package is used to call Wappalyzer directly from Python. The server responses are extracted from the PCAP using *scapy* package.
- **Statistical analysis**
  HTTP requests and responses are extracted from PCAP using *scapy*. Python's *collections.Counter* is used for counting endpoints, HTTP status codes and content types.
- **Database canary**
  The canaries are stored in the database, along with analysis results. Server responses are extracted from the PCAP using *scapy*. Regular expressions are used for identifying the canaries in server responses.

- **Connections**
  The connection graph is retrieved from Moloch API (*/connections.json* endpoint) using *requests* Python package. It is filtered by source IP address. The received graph is visualized in the web interface using *vis.js* library.

4.5 Web interface and Deployment

The web interface is implemented along the API (they are using the same application object). Similarly, it uses Flask for HTTP routing. The user interface is designed using a free template called *Gentelella Admin* [Sil16]. This template is based on *Bootstrap* – a modern CSS framework. It contains pre-defined HTML classes that can be applied to HTML elements and CSS grid used to position elements on the page. Gentelella Admin contains many popular Javascript libraries for graphing, animations *etc.* The content from the database is displayed partially using *Jinja2* templating engine and partially using *Angular* framework using *AJAX*.

The new analysis can also be created through the web interface. It provides a simple form to provide details about the analysis (source IP, destination IP *etc.*) and an option to upload the custom PCAP. The form is mimicking the working of an alert forwarder. If the custom PCAP is provided, it is used instead of a PCAP retrieved from the Moloch. This approach allows using the analysis engine for hosts, that are not monitored by Snort, Moloch or both.

The Flask application has to run under control of some application server. While Flask contains a lightweight web server (used mainly for debugging), it is not suitable for production deployment. There are several application servers that can be used with Flask [Ron16]. I decided to use *uWSGI* [uWS16]. uWSGI is an application server with pluggable architecture, used for deploying *WSGI* applications. WSGI is a standard interface for Python web applications and web servers. Although uWSGI is an application server, it does not contain *Init script* (e.g. for *upstart* init daemon) which is used to start daemons during system boot. uWSGI is by default started in the foreground as a standard process. While it can be switched to the background and managed by Unix signals, I chose to use *supervisor* [Sup16] as a process control system for several processes. The supervisor is used to starting and managing processes which are running in the foreground. The managing part takes care of restarting the processes when needed, logging and much more. Three processes are running under supervisor's control:

- **uWSGI** with API and web interface
- **Celery worker**
- **Alert forwarder**

All deployment scripts are provided in the attached archive (located in *install* directory).

5 Evaluation

To test the correctness of the analysis, I installed all subsystems on a virtual machine with *Ubuntu 14.04. Damn Vulnerable Web App (DVWA)* was used as a target web application. It is a web application developed in PHP, which uses MySQL as a database server [DVW]. DVWA contains many vulnerabilities (by design) used mainly for learning purposes. It can also be used as a testing sandbox.

After discussion with my advisor and consultant, we decided to use sqlmap for testing. While there are other automated tools for SQL injection exploitation (e.g. *sqlchop*, *Havij*), sqlmap provides a large set of options and detailed documentation. Although general web application vulnerability scanners like *acunetix* or *w3af* can detect SQL injection vulnerabilities, they usually lack a functionality to exploit them. More specifically, w3af uses sqlmap when actual exploitation needs to be tested.

5.1 Sqlmap

Sqlmap is a advanced command-line tool used for SQL injection. It has an ability automatically verify the SQL injection, detect the underlying database server and use an appropriate SQL injection techniques (presented in *section 2.5*) to exploit the vulnerability. Sqlmap's workflow is broken into five phases:

1. **Setup**
   Although sqlmap is meant to be fully autonomous, it needs some configuration details before running. The options that are often specified include:
   - URL of the target with entry point specified
   - Additional HTTP headers (Cookies, Authentication *etc.*)
   - Proxy / Tor
   - Evasion (anti-WAF scripts, risk level *etc.*)

   Sqlmap also contains a *wizard* option, which will lead a user through setup phase.

2. **Detection**
   In this phase, sqlmap tests various techniques against the entry point. Sqlmap uses several heuristic methods to detect whether injection technique was successful.

3. **Fingerprinting**
   Using SQL injection techniques and HTTP headers, sqlmap can fingerprint both database server and web server. Retrieved details from this phase are crucial to the success of next two phases. Sqlmap can also detect the WAF or IPS placed before the web server.

4. **Enumeration**
   Sqlmap retrieves a list of databases, tables, columns and content of the tables. The content is usually retrieved from information_schema database, but sqlmap also supports brute-forcing tables and columns name. It is also able to recognize the password hashes and perform hash cracking using a dictionary attack.

5. **Takeover**
   After successful enumeration, sqlmap tries to obtain access to the underlying operating system. Guimarães [Gui09] presents the advanced exploitation techniques in his paper.

Figure 5.1: Command-line interface of *sqlmap*

During testing, I run 23 sqlmap (version **1.0.5.27**) attacks against DVWA. In all cases, Snort was able to detect the SQL injection attempt, generated an alert and created the analysis with proper PCAP. For attacks where database enumeration was performed, the canary module successfully detected the canary string in server's response. Unfortunately, DVWA does not return a 404 error for suspicious inputs, so statistics module was not able to classify it as malicious. Snort generated approximately 40 alerts for each sqlmap attack performing database enumeration. Example sqlmap arguments used for testing:

```
python sqlmap.py -u
"http://[hostname]/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit#"
--cookie "PHPSESSID=[session_id]; security=low"
--dbs --flush-session --tamper=randomcomments --level=5 -v 3
```

Sqlmap also contains functionality to evade IPS/WAF detection using *tamper scripts*. It works by changing the syntax of the input (e.g. inserting escape characters, inserting comments) but preserving its semantics. In most cases *(46/47)* Snort was able to detect such evasion. The *base64encode* is the only tamper script that Snort was not able to detect. However, sqlmap could not retrieve the database content in this case.

6 Conclusion

The primary goal of this thesis was to design and implement the network-based system for automated investigation of SQL injection attacks. The system is designed for the large networks (e.g. enterprise networks) where several tens or hundreds of web servers are deployed. Indeed, to gain the best evidence about the successful injection, direct access to the affected server is usually needed. However, network security devices are still required for additional security. A lot of interesting information can be extracted from network traffic.

The analysis is accomplished by collecting the network traffic for the affected web server. Snort and Moloch were chosen to collect the network evidence. The system was designed and implemented to extract patterns and information from the network traffic using custom modules. Because of the modular architecture, it is pretty simple to create new modules or even use the system for another web attacks investigation (e.g. *cross site scripting*). Although the implemented system is not using any new technique for detecting SQL injection attacks, it automatically correlates information from various systems which saves a lot of time during investigations.

The implemented system has shown itself to be a powerful mechanism for SQL injection analysis. I plan to publish the implemented system on GitHub, so that other developers and security professionals can use and extend it.

6.1 Future plans

Signature-based SQL injection detection has some limitations. The attackers are constantly creating new SQL injection techniques and tools. The IDS cannot include specific signatures for every possible SQL injection query. It is understandable that IDS often has the large ratio of *false negative* classifications.

To cope with this issue, an anomaly-based IDS or a WAF can be used as a primary detection mechanism. However, as presented in *subsection 2.8.2*, only a small number of actual implementations of anomaly-based IDSs are available. Although WAFs have better ability to detect web attacks, they usually lack flexible deployment options (see *subsection 2.8.3*).

A ModSecurity team is working on a *ModSecurity version 3*. Although there is not any official announcement for this release, I had communicated with several ModSecurity developers who confirmed it. The new version comes with a completely new architecture which will allow replacing the IDS as a primary the detection subsystem. A false negative ratio of the detection subsystem will decrease significantly because it will be possible to use WAF engine in an out-of-band deployment scheme.

I would also like to export the analysis results to the *Big Data* solution like *Splunk*. These systems provide a wide range of possibilities for post-processing of the analysis results (e.g. visualizations or filtering).

Bibliography

[Abl+14] L. Ablon *et al.*, *Markets for Cybercrime Tools and Stolen Data*. RAND Corporation, 2014, ISBN: 9780833087119.

[BCS15] Blue Coat Systems, Inc. (2014). Top 10 reasons to deploy Blue Coat ProxySG inconjunction with Next Gen Firewall technology, [Online]. Available: https://www.bluecoat.com/documents/download/86c3778f-7598-43f8-9120-d9e8355fdf31/8a2e583a-76a4-4f48-9a4d-0f76cfe74989 (visited on 05/19/2016).

[Bro] The Bro Project, *Bro introduction*. [Online]. Available: https://www.bro.org/sphinx/intro/index.html (visited on 05/19/2016).

[Cla12] J. Clarke, *SQL Injection Attacks and Defense, Second Edition*. Syngress, 2012, ISBN: 9781597499637.

[DH12] S. Davidoff and J. Ham, *Network Forensics: Tracking Hackers through Cyberspace*. Prentice Hall, 2012, ISBN: 9780132564717.

[Die15] N. Dietrich. (2015). Snort 2.9.8.x on Ubuntu 12, 14, and 15, [Online]. Available: https://s3.amazonaws.com/snort-org-site/production/document_files/files/000/000/090/original/Snort_2.9.8.x_on_Ubuntu_12-14-15.pdf (visited on 05/19/2016).

[DMS04] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router", DTIC Document, Tech. Rep., 2004.

[DVW] RandomStorm, *Damn Vulnerable Web Application (DVWA)*. [Online]. Available: http://www.dvwa.co.uk/ (visited on 05/19/2016).

[EES10] M. Elhamahmy, H. N. Elmahdy, and I. A. Saroit, "A new approach for evaluating intrusion detection system", *Artificial Intelligent Systems and Machine Learning*, vol. 2, no. 11, 2010.

[Fon12] J. Fontana. (2012). Breach clean-up cost LinkedIn nearly $1 million, another $2-3 million in upgrades, [Online]. Available: http://www.zdnet.com/article/breach-clean-up-cost-linkedin-nearly-1-million-another-2-3-million-in-upgrades/ (visited on 05/19/2016).

[For98] J. Forristal. (1998). NT Web Technology Vulnerabilities, [Online]. Available: http://phrack.org/issues/54/8.html#article (visited on 05/19/2016).

[FS11] K. R. Fall and W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols (2nd Edition)*. Addison-Wesley Professional, 2011, ISBN: 9780321336316.

[Gar+09] P. Garcia-Teodoro *et al.*, "Anomaly-based network intrusion detection: Techniques, systems and challenges", *Computers & security*, vol. 28, 2009.

[GC04] F. Guo and T.-c. Chiueh, "Traffic analysis: From stateful firewall to network intrusion detection system", *RPE Report*, 2004.

[GL05] M. G. Gouda and A. X. Liu, "A model of stateful firewalls and its properties", in *In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN-05*, 2005, pp. 320–327.

[Gon12] M. Goncharov, “Russian underground 101”, 2012.

[Goo15] S. Gooding. (2015). Blind SQL Injection Vulnerability Discovered in WordPress SEO Plugin by Yoast, [Online]. Available: http://wptavern.com/blind-sql-injection-vulnerability-discovered-in-wordpress-seo-plugin-by-yoast-immediate-update-recommended (visited on 05/19/2016).

[Gui09] B. Guimarães, “Advanced SQL injection to operating system full control”, Black Hat Europe 2009 conference, 2009.

[Hak] Arkoon Network Security, *Haka*. [Online]. Available: http://www.haka-security.org/ (visited on 05/19/2016).

[Hol+08] T. Holz *et al.*, “Measuring and Detecting Fast-Flux service networks.”, in *Network and Distributed System Security Symposium (NDSS)*, 2008.

[Hor] C. Horsley, *Chorsley/python-Wappalyzer: Python driver for Wappalyzer, a web application detection utility*. [Online]. Available: https://github.com/chorsley/python-Wappalyzer (visited on 05/19/2016).

[Hun13] T. Hunt. (2013). Everything you wanted to know about SQL injection (but were afraid to ask), [Online]. Available: https://www.troyhunt.com/everything-you-wanted-to-know-about-sql/ (visited on 05/19/2016).

[Ish16] J. Ish. (2016). Idstools Documentation: Release 0.5.2, [Online]. Available: https://media.readthedocs.org/pdf/idstools/0.5.2/idstools.pdf (visited on 05/19/2016).

[Kie+09] A. Kieyzun *et al.*, “Automatic creation of SQL injection and cross-site scripting attacks”, in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, IEEE, 2009.

[Kum13] M. Kumar. (2013). Hacker stole $100,000 from users of California based ISP using SQL Injection, [Online]. Available: http://thehackernews.com/2013/10/hacker-stole-100000-from-users-of.html (visited on 05/19/2016).

[Lan16] J.-P. Lang. (2016). Suricata rules, [Online]. Available: https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata_Rules (visited on 05/19/2016).

[LE07] B. Livshits and Ú. Erlingsson, “Using web application construction frameworks to protect against code injection attacks”, in *Proceedings of the 2007 workshop on Programming languages and analysis for security*, ACM, 2007.

[Man15] Mandiant, “M-trends 2015: A view from the front lines”, Mandiant, Tech. Rep., 2015.

[Mey08] R. Meyer, “Detecting attacks on web applications from log files”, SANS Institute, Tech. Rep., 2008.

[Mil11] L. C. Miller, *Next Generation Firewalls for Dummies*. Wiley Publishing Inc., 2011, ISBN: 9780470939550.

[Mol] AOL, *Aol/moloch*. [Online]. Available: https://github.com/aol/moloch (visited on 05/19/2016).

[Nel13] D. Nelson, "Next gen web architecture for the cloud era", Saturn 2013 conference, 2013.

[ONe08] E. J. O'Neil, "Object/Relational Mapping 2008: Hibernate and the entity data model", in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ACM, 2008.

[OWA13] OWASP Foundation. (2013). Owasp Top 10 - 2013, [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10 (visited on 05/19/2016).

[PAN15] Palo Alto Networks Inc., "Palo Alto Networks Next-Generation firewall overview", Tech. Rep., 2015.

[Pap08] N. Pappas, "Network IDS & IPS deployment strategies", SANS Institute, Tech. Rep., 2008.

[PN98] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection", DTIC Document, Tech. Rep., 1998.

[Pri16] M. Prince. (2016). The Trouble with Tor, [Online]. Available: https://blog.cloudflare.com/the-trouble-with-tor/ (visited on 05/19/2016).

[Pro] Proofpoint Inc., *Threat intelligence*. [Online]. Available: https://www.proofpoint.com/us/threat-intelligence-overview (visited on 05/19/2016).

[Reg15] E. Regalado, "Website defense in depth – casting Akamai, Imperva, CloudFlare, F5 and Distil networks in their starring roles", 2015.

[Ret] RethinkDB, *RethinkDB: The open-source database for the realtime web*. [Online]. Available: https://www.rethinkdb.com/ (visited on 05/19/2016).

[Ris10] I. Ristic, *ModSecurity Handbook: The Complete Guide to the Popular Open Source Web Application Firewall*. Feisty Duck Limited, 2010, ISBN: 9781907117022.

[Rob13] C. Roberts, "Discovering security events of interest using Splunk", SANS Institute, Tech. Rep., 2013.

[Roe99] M. Roesch, "Snort - lightweight intrusion detection for networks", in *Proceedings of the 13th USENIX Conference on System Administration*, 1999, pp. 229–238.

[Ron16] A. Ronacher. (2016). Flask Documentation: Release 0.10.1-20160129, [Online]. Available: http://flask.pocoo.org/docs/0.10/.latex/Flask.pdf (visited on 05/19/2016).

[Sch04] F. B. Schneider, "Least privilege and more", in *Computer Systems*, Springer, 2004, pp. 253–258.

[SH09] K. A. Scarfone and P. Hoffman, "Sp 800-41 rev. 1. guidelines on firewalls and firewall policy", Tech. Rep., 2009.

[Sil16] A. Silkalns. (2016). Puikinsh/gentelella: Free bootstrap 3 admin template, [Online]. Available: https://github.com/puikinsh/gentelella (visited on 05/19/2016).

[SM07] K. A. Scarfone and P. M. Mell, "Sp 800-94. guide to intrusion detection and prevention systems (idps)", Tech. Rep., 2007.

[Sol+16] A. Solem *et al.* (2016). Celery Documentation: Release 3.1.23, [Online]. Available: https://media.readthedocs.org/pdf/celery/latest/celery.pdf (visited on 05/19/2016).

[SP10] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection", in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 305–316.

[SP11] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition*. Wiley, 2011, ISBN: 9781118026472.

[Sup16] Supervisor Developers. (2016). supervisor Documentation: Release 4.0.0.dev0, [Online]. Available: https://media.readthedocs.org/pdf/supervisor/latest/supervisor.pdf (visited on 05/19/2016).

[Tal] Cisco Systems Inc., *Talos - snort.org*. [Online]. Available: https://www.snort.org/talos (visited on 05/19/2016).

[Tru15] Trustwave, "2015 Trustwave Global Security Report", Trustwave, Tech. Rep., 2015.

[Uni] Cisco Systems Inc., *Readme.unified2*. [Online]. Available: https://www.snort.org/faq/readme-unified2 (visited on 05/19/2016).

[uWS16] uWSGI. (2016). uWSGI Documentation: Release 2.0, [Online]. Available: https://media.readthedocs.org/pdf/uwsgi-docs/latest/uwsgi-docs.pdf (visited on 05/19/2016).

[Wap] E. Alias, *Wappalyzer*. [Online]. Available: https://wappalyzer.com/ (visited on 05/19/2016).

[Wil16] J. Williams. (2016). Injection theory, [Online]. Available: https://www.owasp.org/index.php/Injection_Theory (visited on 05/19/2016).

[Wir] Wireshark, *Ethernet capture setup*. [Online]. Available: https://wiki.wireshark.org/CaptureSetup/Ethernet (visited on 05/19/2016).

A Web interface screenshots

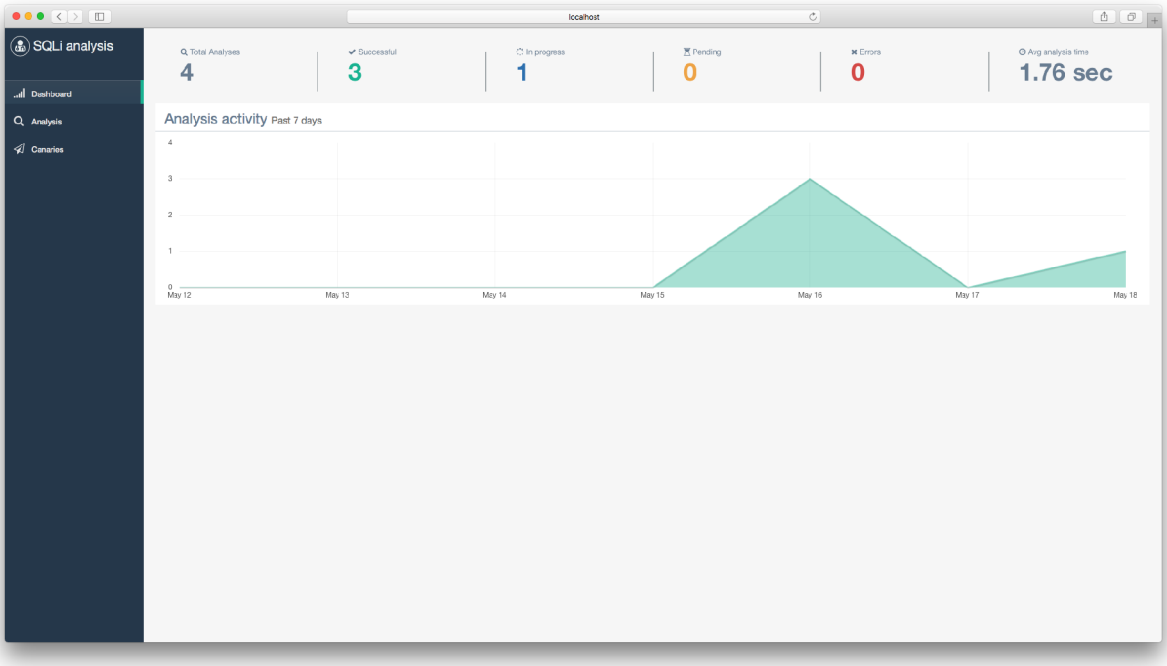

Figure A.1: Dashboards

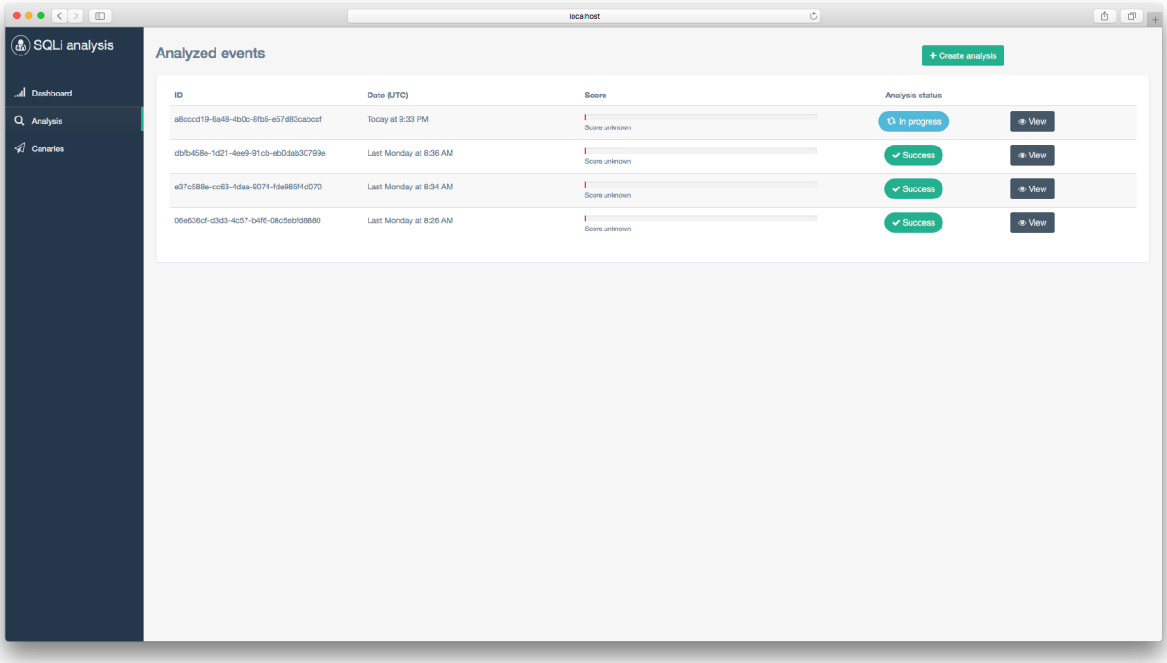

Figure A.2: List of analyses

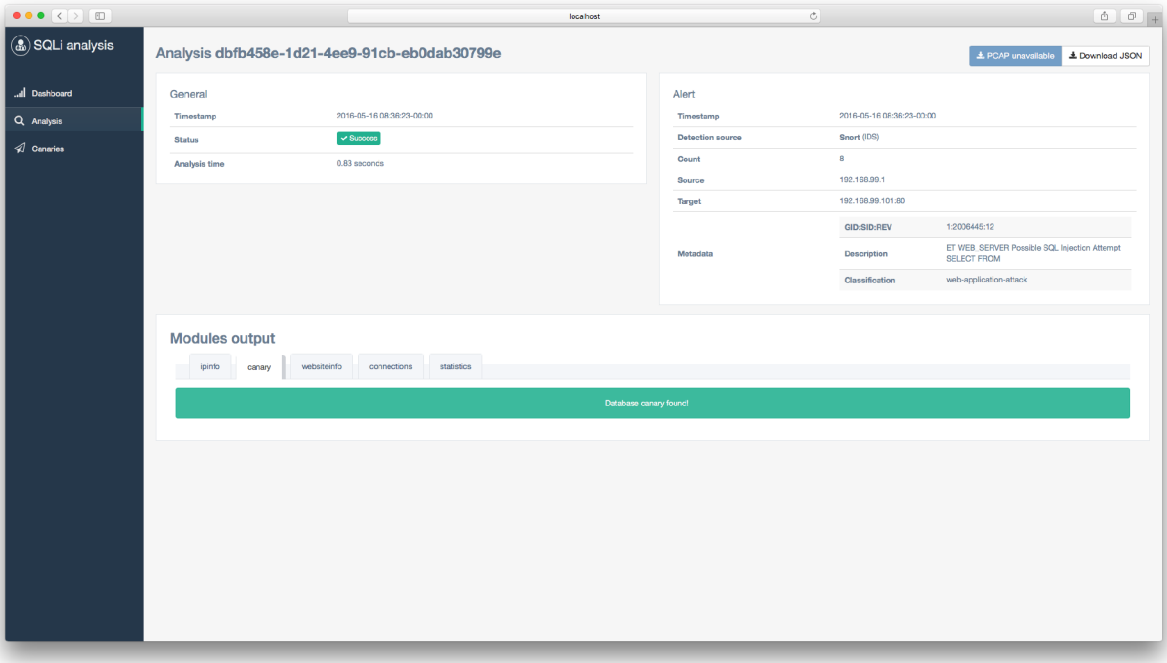

Figure A.3: Analysis result

B API documentation

GET /api/dashboard

Returns data for dashboards.

Example request:

```
GET /api/dashboard HTTP/1.1
Host: example.com
Accept: */*
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "analysis_count": 3,
  "analysis_graph": [
    [
      1462838400000,
      0
    ],
    "...Abbreviated for clarity..."
  ],
  "error_count": 0,
  "exec_time": 1.76,
  "pending_count": 0,
  "progress_count": 0,
  "success_count": 3
}
```

GET /api/analysis

Returns list of all analyses with basic details.

Example request:

```
GET /api/analysis HTTP/1.1
Host: example.com
Accept: */*
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "analyses": [{
    "date": "2016-05-16T08:36:23.929000+00:00",
    "id": "dbfb458e-1d21-4ee9-91cb-eb0dab30799e",
    "job_status": {
      "state": "SUCCESS"
    }
  }]
}
```

GET /api/analysis/<task_id>

Returns analysis details for analysis with id of *task_id*.

Example request:

```
GET /api/analysis/dbfb458e-1d21-4ee9-91cb-eb0dab30799e HTTP/1.1
Host: example.com
Accept: */*
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "alert": [{
    "dest_ip": "192.168.99.101",
    "dest_port": 80,
    "detection": "snort",
    "meta": {
      "cid": 28,
      "classification": "web-application-attack",
      "description": "ET WEB_SERVER Possible SQL Injection...",
      "gid": 1,
      "reference": null,
      "rev": 12,
      "sid": 2006445
    },
    "src_ip": "192.168.99.1",
    "src_port": 55871,
    "timestamp": 1463387783
  }],
  "date": "Mon, 16 May 2016 08:36:23 GMT",
  "exec_time": 0.83,
  "id": "dbfb458e-1d21-4ee9-91cb-eb0dab30799e",
  "job_freeze": 60,
  "job_start": "Mon, 16 May 2016 08:37:23 GMT",
  "job_status": {
    "state": "SUCCESS"
  },
  "pcap_file": "192-168-99-1.1463387783.pcap",
  "results": {
    "canary": {
      "found": true
    }
  }
}
```

POST /api/analysis

Processes the alert. It will either create a new analysis or aggregate the alert to the existing analysis. Analysis ID is returned.

Example request:

```
POST /api/analysis HTTP/1.1
Host: example.com
Accept: */*

{
  "dest_ip": "192.168.99.101",
  "dest_port": 80,
  "detection": "snort",
  "meta": {
    "cid": 28,
    "classification": "web-application-attack",
    "description": "ET WEB_SERVER Possible SQL Injection...",
    "gid": 1,
    "reference": null,
    "rev": 12,
    "sid": 2006445
  },
  "src_ip": "192.168.99.1",
  "src_port": 59582,
  "timestamp": 1463443866
}
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "job_id": "a8cccd19-6a48-4b0c-8fb5-e57d83cabccf"
}
```

GET /api/pcap/<pcap_filename>

Returns PCAP with *pcap_filename* filename.

Example request:

```
GET /api/pcap/10-102-17-243.1457694721.pcap HTTP/1.1
Host: example.com
Accept: */*
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/vnd.tcpdump.pcap

#BINARY DATA#
```

GET /api/canary

Returns list of generated canaries.

Example request:

```
GET /api/canary HTTP/1.1
Host: example.com
Accept: */*
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "canaries": [{
    "canary": "bada50878b1ed4e27a99dc6c590f661b3...",
    "date": "Mon, 16 May 2016 08:33:27 GMT",
    "id": "a1c3db47-5a18-4d13-bfcf-f5b5b8ef455e"
  }]
}
```

POST /api/canary

Generates new canary string.

Example request:

```
POST /api/canary HTTP/1.1
Host: example.com
Accept: */*
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "canary": "0a32325458c1f7f7f4e3c96b87fbd7234b52..."
}
```

POST /api/canary/<canary_string>

Deletes *canary_string* canary.

Example request:

```
POST /api/canary/0a32325458c1f7f7f4e3c96b87fbd7234b52... HTTP/1.1
Host: example.com
Accept: */*
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
    "status": "ok"
}
```