BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

HTTP APPLICATION ANOMALY DETECTION

DETEKCE ANOMÁLIÍ HTTP APLIKACÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VLASTIMIL RÁDSETOULAL**
AUTOR PRÁCE

SUPERVISOR **Mgr. Ing. PAVEL OČENÁŠEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Rádsetoulal Vlastimil**
Program: Informační technologie
Název: **Detekce anomálií HTTP aplikací**
HTTP Application Anomaly Detection
Kategorie: Web

Zadání:

1. Seznamte se s principy analýzy anomálií v prostředí systémů počítačových sítí.
2. Analyzujte požadavky na systém umožňující analýzu HTTP provozu a modelování standardního a detekci nestandardního chování aplikací (např. u přechodů mezi stránkami apod.)
3. Navrhněte systém pro detekci anomálií dle předchozího bodu a dle instrukcí vedoucího práce.
4. Navržený systém implementujte.
5. Implementovaný systém ověřte vhodně zvolených na reálných datech.
6. Diskutujte získané výsledky a možnosti dalšího rozšíření.

Literatura:

- Kurose, J. F. Computer networking: A top-down approach. Pearson, Essex, 2017, ISBN 978-1-292-15359-9.
- Stallings, W. Network security essentials: Applications and standards. Hoboken, 2016, ISBN 978-0-13-452733-8.
- Bishop, M. Computer security: Art & Science. Addison-Wesley, Boston, 2003, ISBN 0-201-44099-7.
- Buczak, A., Guven, E.. A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. IEEE Communications surveys and tutorials. IEEE, 2016, 18(2), s. 1153-1176.
- Kruegel, Ch., Vigna, G. Anomaly Detection of Web-based Attacks. In: Proceedings of the ACM Conference on Computer and Communications Security. ACM, Washington, DC, USA. 2003.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Očenášek Pavel, Mgr. Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 27. října 2020

Abstract

The goal of this work is to introduce anomaly detection principles and review its possibilities, as one of the intrusion detection methods in HTTP traffic. This work contains theoretical background crucial for performing an anomaly detection on HTTP traffic, and for utilising neural networks in achieving this goal. The work proposes tailored design of an anomaly detection model for concrete web server implementation, describes its implementation and evaluates the results. The result of this work is successful initial experiment, of modeling normal behavior of HTTP traffic and creation of the mechanism, capable of detection of anomalies within future traffic.

Abstrakt

Cieľom tejto práce je predstaviť princípy a odhaliť možnosti detekcie anomálií v HTTP prevádzke, ako jednej z metód, pre detekciu pokusov o prienik do webových systémov. Táto práca obsahuje teoretický základ, kritický pre detekciu anomálií v HTTP prevádzke a pre využitie neurónových sietí, k jej implementácii. Práca predstavuje dizajn modelu pre detekciu anomálií, ušitý na mieru pre konkrétny webový server v tejto práci, opisuje jeho implementáciu a hodnotí výsledky. Výsledok tejto práce je úspešný prvotný experiment, ktorý spočíva v modelovaní bežnej, neškodnej HTTP prevádzky a vytvorení mechanizmu, ktorý je schopný detegovať anomálie v budúcej prevádzke.

Keywords

anomaly, detection, autoencoders, HTTP, neural, networks

Kľúčové slová

anomália, detekcia, auto-enkóder, HTTP, neurónové, siete

Reference

RÁDSETOULAL, Vlastimil. *HTTP Application Anomaly Detection.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Ing. Pavel Očenášek, Ph.D.

Rozšírený abstrakt

Cieľom tejto práce je predstaviť princípy a odhaliť možnosti detekcie anomálií v HTTP prevádzke, ako jednej z metód, pre detekciu pokusov o prienik do webových systémov. Táto práca obsahuje teoretický základ, kritický pre detekciu anomálií v HTTP prevádzke a pre využitie neurónových sietí, k jej implementácii. Na to je potrebné porozumieť základným princípom fungovania HTTP protokolu. Práca predstavuje základné súčasti tohoto protokolu, akými napríklad sú HTTP požiadavky a odpovede, alebo jednotlivé hlavičkové polia HTTP požiadaviek. Ďalšia časť kapitoly o HTTP vysvetľuje, kde sa protokol nachádza, z pohľadu modelovania sietí pomocou sieťových modelov. Základné bezpečnostné riziká a implementačné zraniteľnosti, sú vysvetlené na konci tejto kapitoly, z toho niektoré, vybrané, sú vysvetlené podrobnejšie. Ako možnosť využitia neurónových sietí, pre účely detekcie anomálií, je v práci predstavená neurónová sieť, v podobe auto-enkódera. Implementačná časť je naprogramovaná v jazyku Python, ako široko používaným programovacím jazykom, pre vedecké účely. Pre účely modelovania neurónových sietí a ich následného spustenia, je využitý nástroj TensorFlow, ktorý je predstavený v kapitole o detekcii anomálií. Práca predstavuje dizajn riešenia pre detekciu anomálií, ušitý na mieru, pre konkrétny webový server, ktorého HTTP prevádzka bola nasimulovaná spoločnosťou GREYCORTEX s.r.o., počas penetračného testovania implementácie tohoto servera. Táto prevádzka je nutne rozdelená do dvoch častí, na neškodnú prevádzku a na prevádzku, ktorá obsahuje HTTP požiadavky, prichádzajúce serveru počas útokov na neho. K implementácií je použitý už vyššie spomenutý auto-enkóder, ktorý sa natrénuje pomocou spracovaných častí požiadavok neškodnej prevádzky, v podobe URI. Detekcia anomálií potom spočíva v tom, že tento model by mal byť schopný s určitou presnosťou skopírovať svoj vstup, na svoj výstup. Keďže bol tento model natrénovaný s pomocou dát z neškodnej prevádzky, dáta zo škodlivej prevádzky nebude vedieť zrekonštruovať a vyprodukuje rekonštrukčnú chybu, vyššiu ako je stanovený limit, pre určenie anomálie. Táto chyba sa potom porovná s hraničnou hodnotou, ktorá sa stanoví pomocou súčtu priemeru vypočítaného z rekonštrukčných chýb normálnej prevádzky a experimentálne získanej fixnej hodnoty. Táto hodnota býva zvyčajne tri smerodatné odchylky distribúcie dát. Práca opisuje implementáciu navrhnutého modelu a hodnotí výsledky a výstupy, dosiahnuté experimentami. Výsledok tejto práce je úspešný, prvotný experiment, ktorý spočíva v modelovaní bežnej, neškodnej HTTP prevádzky a vytvorení mechanizmu, ktorý je schopný detekovať anomálie v budúcej prevádzke. Na záver práce sú zhodnotené výsledky práce a navrhnuté možné implementačné zlepšenia, vrátane nápadov pre budúce návrhy, oveľa komplexnejších systémov pre detekciu anomálií, ktoré by mohli byť schopné, dynamicky vyhodnocovať HTTP požiadavky, alebo presnejšie ich dávky, v reálnej prevádzke.

HTTP Application Anomaly Detection

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mgr.Ing.Pavel Očenášek, Ph.D. The supplementary information was provided by Ing.Peter Chmelář, Ph.D and Ing.Marina Volkova, Ph.D from GREYCORTEX s.r.o. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Vlastimil Rádsetoulal
May 12, 2021

Acknowledgements

Hereby I would like to express my sincere appreciation for the help of my supervisors Mgr.Ing.Pavel Očenášek, Ph.D., for the initial consultations and peaceful work, and Ing.Peter Chmelář, Ph.D. for his always optimistic attitude during online consultations and all the information and help provided. I would like to express the same level of appreciation for the help and information from Ing.Marina Volkova, Ph.D.

Contents

Chapter 1

Introduction

The importance of internet security implementations is raising year by year, as the internet and the world-wide web became an indisputable parts of our everyday lives. Malicious actors from all over the world, are constantly attempting to take an advantage of different vulnerabilities within web application implementations to steal sensitive data, interrupt services, generate an income and more.

In pursuit of securing the networks of organisations and their web systems, there are multiple layers of the protections implemented. From firewalls and secure configurations, to user access control, to malware protection or patch management. One of the networking elements that needs to be protected from various types of attacks is the one, that the end users come in contact with the most. It is the application layer, more specifically web applications on HTTP. As this applications come in contact with the world, connected to the internet, their security robustness is critical. One of the methods to prevent adversaries from achieving their goals, is intrusion detection. In this work we will try to detect malicious incoming HTTP messages. For this purpose we will be looking at potentially malicious, abnormal HTTP requests and we will talk about them as anomalies. An anomaly stands for deviation from norm. In order to define such norm, we will be modeling normal behavior of HTTP traffic. We will be exploring the possibilities of neural networks, concretely autoencoders, in reaching our goal. We will try to model norm of HTTP web server traffic, provided by GREYCORTEX s.r.o., from which has this assignment landed. After successful modeling of the norm, we can perform anomaly detection, that should detect abnormal HTTP requests, in comparison to this norm.

In this journey we will need to introduce basic components of HTTP, the version used in mentioned traffic, where does HTTP stand in terms of networking models and some of the critical vulnerabilities and risks in chapter 2. Then to understand anomaly detection problematic, to understand neural networks in form of autoencoders and to introduce the powerful tool TensorFlow, was chapter 3 created. The chapters 4 and 5 describe the design, data preprocessing needed, implementation parts and results of our concrete experiment.

Chapter 2

Hypertext Transfer Protocol

In this chapter we will discuss Hypertext Transfer protocol's brief version history and its version evolution. We will then go more in depth examining its messages and also explaining its position on OSI model. At the end we will have a look at some known attack possibilities.

2.1 HTTP and its Versions

According to [5] The Hypertext Transfer Protocol is an application-level protocol that provides lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol suitable for many tasks, through extension of its request methods. A feature of HTTP is the typing of data representation, which allows systems to be built independently of the data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990. HTTP is a request/response protocol in which HTTP client sends requests to the server listening for TCP [20] connections, usually on ports 80 and 443, depending on whether secure version of protocol is used or not. However ports may vary depending on server implementation.

The first version of HTTP denoted as HTTP/0.9 is very simple where requests consist of one line and was used for transfer of raw data across the internet. This version has not been standardized and initially had no version number.

In terms of standardization the first version to be officially defined in RFC document is HTTP/1.0. It is defined and described in RFC1945. In this protocol messages are allowed to be in MIME-like format and contain meta-information about transferred data and other modifiers in requests and responses [13]. However first documented version of protocol is lacking consideration of some important technical aspects such as caching, proxies or persistent connections. Counting all these reasons and some more there was a need for protocol version change, which resulted in defining HTTP/1.1 in RFC2616. This RFC has then been obsoleted by series of RFC documents RFC7230-RFC7235.

Despite the fact, that HTTP/2 provided as standard in [3] is the latest version of HTTP and HTTP/3 is in development, for now as an internet draft, we will explain general information about HTTP from the series of RFC documents about HTTP/1.1. The reason for this is also the fact, that provided HTTP traffic of the web server in this work is in this protocol version.

2.2 HTTP on OSI

In computer networks it comes to great importance harmony between hardware and software elements. For easier understanding of complicated network architectures, the networks are described divided into parts. This leads to creation of layered models where these parts are layers interconnected rather functionally than physically [1]. As one of the reasons for using layered models to describe computer networks is to simplify understanding of the network model, it is crucial to know on which layer do we operate when working with networking elements. We chose to describe ISO OSI model, due to its higher granularity than in case of TCP/IP, leading to greater understanding of individual steps taken in modeling network services. This section will briefly lead us in knowing where HTTP is and for that the information from [1] is going to serve us.

2.2.1 Layered Network Models

There are several reasons to use the description of computer networks in form of layered models summarized and presented in [1]:

- Simplification of understanding the network model.
- The network layering based on functions eases implementation, as the functions of each layer are consistent and distinct. Then programming software implementations and designing hardware based on its functionality within the layer is easier.
- The troubleshooting of the network is able to be separated into troubleshooting of individual layers and thus a potential error can be isolated and corrected within its layer without affecting other network functions.
- The development and implementation of the functions in each layer can be focused on its own duties and the protocols specifically designed for each layer are more efficient, meanwhile the lower layers of the model are maintaining the transparency towards higher layers.

Two main standard layered models are **ISO OSI** and **TCP/IP** pictured in 2.1

2.2.2 ISO OSI Model

The ISO OSI model consists of seven layers shown in the left part of 2.1. Each layer of ISO OSI model handles data in its specific unit called Protocol Data Unit from now on referenced as PDU. Some layers add layer-specific information in form of a header, a trailer, or both to the data. The header is situated at the beginning of the PDU and the trailer information at the end. The information within the header and trailer is used for controlling the communication between two entities on the layer [1].

Network devices such as routers, switches, network interface cards and more usually operate in the bottom three layers and the hosts in the whole seven layers. Let's take a look at each layers brief explanation, but not going too much in depth, as it would be out of the scope of this work [1].

Physical Layer handles data as raw bits thus the PDU for the physical layer is a *bit*. The purpose of this layer is to transparently transmit bits from the data-link layer of the

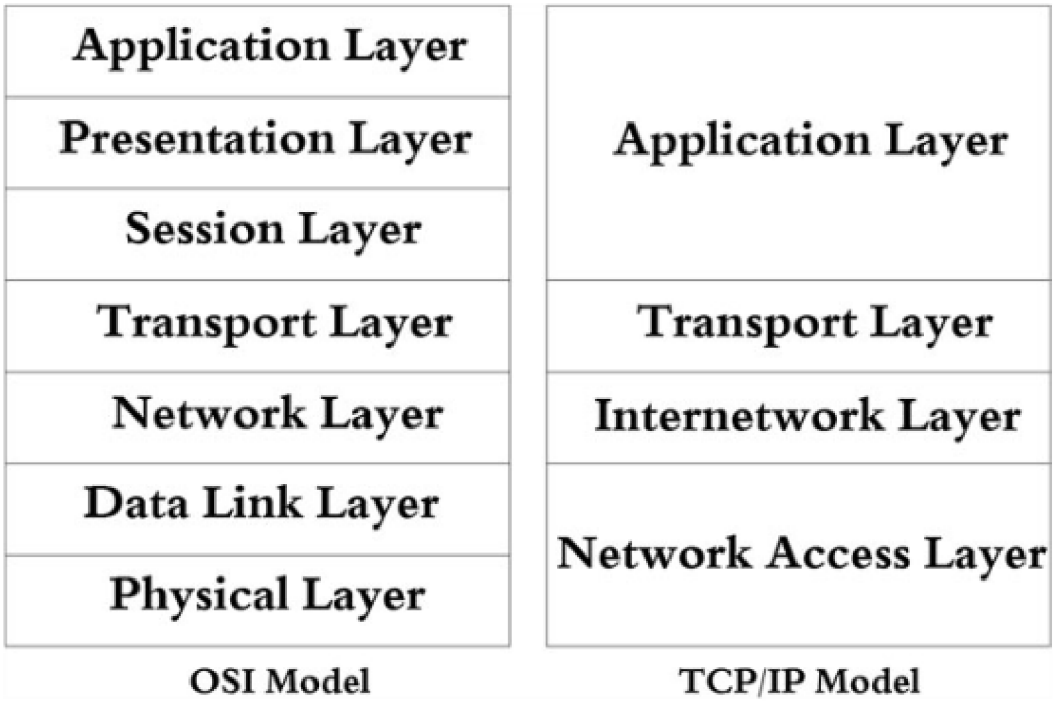

Figure 2.1: OSI and TCP/IP layered models: **left** OSI model, **right** TCP/IP model. Adopted from [1].

sender to the data-link layer of the receiver. This transmission not only includes data, but also additional control information. Physical layer protocols used then vary depending on the type of physical medium and the signal transmitted. The signal can be represented and sent as an electrical voltage in cables, light signals in fiber links, or even through the air in form of electromagnetic signal [1].

Data-Link Layer provides us with PDU in form of a *frame*. It servers the network with multiple functions such as controlling the inter-connections of data-circuits within physical layer, identification and parameter exchange, error in transmission detection, relaying and more [1].

Network Layer has a PDU unit known as a *packet*. This layers is responsible for routing the data from one network and controlling the subnet, relaying. It provides network connections between transport layer entities by utilising the data-link connections available. It handles segmentation and blocking of the packets in case of different data-link standards and packet sizes. It not only detects errors using notifications from the data-link layer and its own detection mechanisms, but can in some cases provide recovery from them. It assures maintaining the sequential order of the packets and controls the flow to prevent flooding its destination with excess data. Network layer is capable of many more tasks, but this is only a brief explanation and is sufficient for our imagination of the layer's work [1].

Transport Layer uses a *segment* as PDU and provides two different types of services depending on whether connection needs to be established or not. These divide into connection-oriented communication and connectionless. The two most common transport protocols used in accomplishing these goals are Transmission Control Protocol (TCP)[20]

and User Datagram Protocol (UDP)[19]. In connection-oriented communication, where the protocol TCP is used, this layer provides additional end-to-end error detection, establishment and release of transport connections, segmentation of the data into segments at the sender and reconstruction at the recipient. It is also capable of monitoring Quality of Service parameters and PDU delimiting, in order to maintain the continuity of communication. Important part of its functionality is also sequence control, for ensuring warranty of the data arrival in unchanged sequence from the sequence initially sent. The transport layer also helps the session layer to differentiate which data belongs to what session [1].

Session Layer unlike other previously mentioned layers does not have its own PDU representation. It handles the data in the form that it is provided. The purpose of the session layer is to support organization of the communication of presentation entities, when multiple simultaneous communication sessions take place. It is responsible for starting the sessions between communicating entities, token management, which serves to identify which entity owns the token for data transmission. Session layer as well provides mapping session connections to transport connections [1].

Presentation Layer serves for the data form negotiation with another communicating entity. After successful negotiation it can provide multiple different services to application layer such as encryption, compression and translation. The application then may choose which services it wishes to use [1].

Application Layer defines the services and functions provided at user end. The protocols used in application layer vary depending on the type of user data to be transferred. The layer defines different acceptable Quality of Service parameters for each service provided. It also decides what security mechanisms should be used, such as access control or authentication and in connection-oriented services the application layer is responsible for synchronization of these services [1].

2.2.3 Where is HTTP

In one sentence HTTP is an application layer protocol. If we take a look at the definition in [11] stating that HTTP is a stateless application-level request/response protocol with extensible semantics and self descriptive message payloads and review its message semantics, we will come to knowing that HTTP does not need take an advantage of any stored context on server. Although HTTP does provide mechanisms for state management in [2] in form of cookies, its concept of session is different from that in OSI model. In terms of presentation layer, some of its functions may be included within HTTP request in form of request header fields, such as content negotiation. Then data encryption is used by HTTP's secure version HTTP over TLS described in [22].

Thus HTTP somehow „touches“ all top three layers of OSI model, but it's functions cannot be strictly assigned by one of them, except for application layer. For simplicity it is much more suitable to consider TCP/IP model pictured in the right part of 2.1, where HTTP would simply belong to application layer. On both layered models HTTP servers use transport layer to listen to the incoming connection's over Transmission Control Protocol(TCP) [20].

```
Layer HTTP:
        POST /firmware.cgi?LD_DEBUG=help HTTP/1.1\r\n
        Expert Info (Chat/Sequence): POST /firmware.cgi?LD_DEBUG=help HTTP/1.1\r\n
        POST /firmware.cgi?LD_DEBUG=help HTTP/1.1\r\n
        Severity level: Chat
        Group: Sequence
        Request Method: POST
        Request URI: /firmware.cgi?LD_DEBUG=help
        Request URI Path: /firmware.cgi
        Request URI Query: LD_DEBUG=help
        Request URI Query Parameter: LD_DEBUG=help
        Request Version: HTTP/1.1
        Host: 147.229.147.168\r\n
        User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)\r\n
        Content-Type: application/x-www-form-urlencoded\r\n
        Content-Length: 0\r\n
        Content length: 0
        Full request URI: http://147.229.147.168/firmware.cgi?LD_DEBUG=help
        HTTP request 1/1
        \r\n
```

Figure 2.2: Example of HTTP request.

2.3 HTTP Messages Overview

Hypertext Transfer Protocol is based on client-server architecture and as we already know message of the protocol is either a request or response. A server listens on a connection waiting for a request, to be able to parse received message. Then its task is to interpret the message semantics and respond to the request related to desired source. A Client on the other hand creates request messages and examines received responses from server. We will be working with HTTP requests in our Anomaly Detection so in this section we will take a look at request messages and possible responses from server. The information in this section is retrieved from [12].

When we retrieve HTTP request from packet capture file we can see it contains multiple fields and its tokens alongside with request message and optionally content data in payload. The figure 2.2 shows us an example of request message and its header fields.

2.3.1 Request Methods

The request method is the indicator of the client's intentions and primary source of the semantics of request. It also defines the expected response by client from server. The standardized methods in HTTP are not specific for the resource and should have the same semantics applied to any resource upon definition. Then it is up to each resource to determine whether the proclaimed semantics are implemented or allowed [12].

The following are standardized methods commonly used in HTTP, defined by [12]:

- GET: method requests to transfer a current representation of the target resource.
- HEAD: method requests to transfer only the status line and header section of the resource.

- POST: used to perform processing that is specific for the resource on the request payload.
- PUT: method requests to create or replace the state of target resource representations with the request payload.
- DELETE: method prompts to remove all current associations between the target resource and its current functionality.
- CONNECT: requires from server to establish a tunnel identified by the target resource.
- OPTIONS: this method serves for the description of communication options for the target resource.
- TRACE: method requests for a remote, application-level loop-back of the request message and must not contain payload data.

It is expected from all general-purpose servers to support methods GET and HEAD, but other methods are optional, depending on implementation [12].

2.3.2 Request Header Fields

Request header fields purpose is to provide more information about the request context, suggest preferred formats for the expected response, provide authentication credentials, modify the request processing or even make the request conditional based on the resource state [12]. Request header fields divide into these groups:

- Controls
- Conditionals
- Content Negotiation
- Authentication Credentials
- Request Context

Controls shown in figure 2.3 are request header fields that are responsible for directing specific handling of the request. For example Expect header field can inform server, that client wishes to send large message body in this request with token „100-continue“. Then client waits for the indication, represented by response code 100, if it is worth sending message body in advance, before actually sending it.

Conditionals request header fields allow client usage of precondition on the state of target source, before execution of the action related to requested source, which has to be decided by this precondition. The evaluation portion of such condition depends on the request method semantics and conditional [10].

Content negotiation header fields sent by user agent help in proactive negotiation about the response content. For example Accept field is used to determine the media types that are acceptable by user agent in the response. Or Accept-Charset field used to help to identify different char set capabilities of user agent [10].

Header Field Name	Defined in...
Cache-Control	Section 5.2 of [RFC7234]
Expect	Section 5.1.1
Host	Section 5.4 of [RFC7230]
Max-Forwards	Section 5.1.2
Pragma	Section 5.4 of [RFC7234]
Range	Section 3.1 of [RFC7233]
TE	Section 4.3 of [RFC7230]

Figure 2.3: Controls request header fields and their definitions in RFC documents. Picture adopted from [12]

Authentication credentials type of header fields contain two header fields responsible for deliverance of authentication credentials, which are alongside authentication further explained and defined in [9]. Two header fields present in this group are Authorization and Proxy-Authorization.

Request context header fields provide some additional information about user agent in field User-Agent. The User-Agent field helps servers to identify range of interoperability problems and avoid user agent limitations. There is also field with name From used for storing contact information, concretely e-mail address of person controlling the requesting user agent [12].

2.3.3 Response Status Codes

In our anomaly detection we will not really take into consideration responses from the server as we are going to detect anomalous requests from clients and the modeled system will not need to include server responses. However, there are ways to use server responses, pairing them with according client requests for anomaly detection. For tnhe completion we can list and describe the main groups from [12]:

- 1xx: responses beginning with number 1 are of Informational character e.g., the request was received or continuing process.
- 2xx: this group of responses are indication of success and may tell us, that the request was received, understood and accepted.
- 3xx: indicates redirection, meaning further action in order to complete the request are needed.
- 4xx: client error response codes that server sends when the request either contains bad syntax or cannot be fulfilled. For example requested resource is not present.
- 5xx: server error response codes signalling the server failure to fulfill an apparently valid request.

2.3.4 HTTP Request URI

There are many different ways for adversaries to exploit HTTP protocol. HTTP request fields are easy to modify for the needs of an attacker. In our anomaly detection we will focus strictly on HTTP request URI.

URI stands for Uniform Resource Identifier which is divided into subsets Uniform Resource Locator and Uniform Resource Name. In HTTP URI identifies the resource for which the request should be applied. URIs used in HTTP are represented in either absolute or relative form [4].

2.4 Attacks on HTTP

In today's world and historically the awareness of security threats across computer networks is on slow rise. On the other hand attackers are faster and willing to exploit any vulnerabilities created in quick development, especially in web applications as they are widely used and often operate with sensitive data. Addressing this problems a nonprofit foundation The Open Web Application Security Project(OWASP) created standard awareness document for developers of web applications and purposes of web application security, about the most critical security risks [14]. We will review this list of top ten critical security risks to web applications and recognize which ones of them can we consider for anomaly detection algorithms.

The following is full list of top ten critical security risks provided by [14]:

- Injection
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities (XXE)
- Broken Access Control
- Security Misconfiguration
- Cross-Site Scripting XSS
- Insecure Deserialization
- Using Components with Known Vulnerabilities
- Insufficient Logging and Monitoring

2.4.1 Injection

Different types of injections such as SQL, NoSQL, OS , or LDAP injection occur, when part of a command or query contain malicious, untrusted data sent to an interpreter. This data can trick the interpreter into accessing, editing or removing data without legitimate authorization or even in executing commands [14].

There are multiple reasons why application could be vulnerable for such attacks. If data supplied by user is not validated, filtered or sanitized before interpreting or dynamic queries or non-parameterized calls are used directly in the interpreter without context-aware escaping. Even when stored procedures are parameterized, they can still introduce SQL injection if PL/SQL or T-SQL processing extensions concatenate queries and data, or executes malicious data with *EXECUTE IMMEDIATE* or *exec()*. The best prevention proposed by OWASP is simply code review followed by thorough testing of all parameters, headers, URL, cookies and different data type inputs processed [14].

In addition to usage of safe APIs, avoiding the use of the interpreter entirely or at least providing parameterized interface, usage of SQL controls within queries, escaping special characters with specific escape syntax given by interpreter for any residual dynamic queries and more, anomaly detection could provide great first contact protection against incoming hostile data. By detecting anomalous portions of queries, cookies or other injection vectors in HTTP traffic even an otherwise vulnerable web-application attacks could be avoided [14].

2.4.2 Broken Authentication and its Automated Attacks

In order to prevent authentication-related attacks, securely implemented confirmation of user identities, authentication, and session management are essential. In case of incorrect implementation of functions responsible for authentication and session management within web applications, several security incidents might take place. From credential theft, sensitive information leaks to compromising the whole system. For example if application does not implement automated threat or credential stuffing protections, attackers are free to use millions of valid username and password combinations for automated injection in order to gain access to accounts. It is also dangerous if system permits usage of insufficiently strong or well-known passwords, such as „12345“ or combination of password and username „admin/admin“. Some other weaknesses could be unencrypted or weakly hashed passwords, exposure of Session IDs in the URL, improper rotation of Session IDs after successful login and their invalidation [14].

There are multiple ways anomaly detection algorithms might be useful in detection of such attacks. In case of credential stuffing and other brute force or automated attacks anomaly detection algorithms would need to be able to recognize unusually large number of authentication attempts and evaluate the traffic as anomalous. It would be important that the anomaly detection mechanism would work not only with single HTTP requests, but with batches of them so it would find if the requests sent to the server had any logical connection [14].

2.4.3 XML External Entities (XXE)

Some applications use or allow usage of the XML format for data transmission between the client and the server. The principle of this vulnerability lies in the fact that many older or poorly configured XML processors evaluate references to external entities within XML documents. These external entities then can serve attackers in disclosing internal files using the file URI handler, internal port scanning, denial of service attacks or even remote code

execution. If applications accept XML directly from untrusted sources and allow input of untrusted data into its XML documents, they are vulnerable to attackers interfering with their internal processing of XML data [14].

An example of such external entity can be Uniform Resource Identifier, that is dereferenced and evaluated in the processing of an XML document.[16]. There is also possibility that when HTTP server usually expects to receive the messages in default forms, such as *Content-Type: application/x-www-form-urlencoded* it can also accept other content types as well as XML. Then if adversaries use XML formatted requests, they can try to exploit XXE vulnerabilities [14].

In the case of uploads, hypothetical anomaly detection algorithm could be able to recognize, that usually web application, that it is working with, is not used to receive data in XML format and mark this attempt as anomalous. Then some security analyst would be able to decide whether some hidden intentions were persuaded [14].

Chapter 3

Anomaly Detection

The main focus of this work is to detect anomalies in HTTP traffic. There are many different approaches and use cases for anomaly detection, but let's first take a look at the definition of anomaly and its detection possibilities. This chapter mainly follows information published in [18].

In one sentence, anomalies are substantial variations from the norm. Anomalies are also called outliers, as these samples or even whole datasets „lie“ in noticeable distance from data, that is considered normal.A Good example to picture anomaly is results of an IQ test.

The usual expected value of one's IQ test results is around 100, with a standard deviation of 15. If someone scores in such test one standard deviation higher or lower, this is not considered an anomaly, despite this result varies from the norm. The result that is around three times standard deviation from the mean, in this case it would be for example 145, but also 65, is to be considered anomalous. This example is simple and only uses single quantitative attribute (IQ score) with an unimodal distribution, from well-known statistics. Most of the problems solved by anomaly detection algorithms are multidimensional, and may involve nominal or categorical variables[18].

In statistics categorical variable places an individual into one of several groups or categories on the basis of some qualitative property [8]. Categorical variables might also be used in our HTTP traffic anomaly detection. For example taking into consideration HTTP method in combination with content-length and request URI could identify some anomalous requests that don't usually occur in previously defined normal traffic. Another possibility would be using User-agent field to help to identify unusual usage of obsolete browsers, with several vulnerabilities that adversaries may take an advantage of. The problem with this approach is, that we would need very large dataset of HTTP requests and still legitimate browsers accessing the server, that are not common or new, would be marked as an anomaly. We will discuss this problem in implementation section of this work.

Anomaly detection principle is based on models and predictions achieved from past data. The statistics used to describe behaviour or characterize a system in the past will continue to characterize behaviour or system in the future [18].

In HTTP traffic we can simulate normal traffic in closed environment using web application or system how it was intended. If the simulation is extensive and varied enough we can expect to see similar traffic in the future. In some real cases, data that changes over

time, for example increasing heights or lifespan among humans, can be characterized by long-term trends, or by cyclic behavior. Changes in structure of web application or system need to be taken into consideration and recreating or adjusting the previous model could become necessary.

3.1 Critical Questions in Anomaly Detection

It is important to ask questions presented in [18] modified for needs of this thesis and answer them, before deciding what approach to take in anomaly detection. These questions are relevant to the formulation of anomaly detection algorithms:

- How is the norm characterized ?
- What to do in case of multiple substantially different cases considered as normal ?
- What is substantial variation in our particular problem ?
- How do we address multi-attribute data ?
- How do we solve changes occurring over time ?

The norm is in our case characterized as HTTP traffic on the server, which is not affected by any misuse or attack. The traffic consists of HTTP requests, achieved by simulation of casual user's behavior in the system. It is important to simulate the traffic in closed environment, as if we used real traffic when server is connected to the internet, there is no affirmation that some otherwise anomalous samples or even blocks of samples are getting into our modeled normal behavior. It is also important to note, that there are attacks which do not look anomalous at first glance. For this type of attacks, anomaly detection is not suitable and should be used in the combination with other intrusion detection techniques.

The structure of HTTP web applications or systems typically does not allow substantially different cases of usage. There might be instances of some less used parts of the system, that could cause minor variation in normal traffic model, however these would not be significant enough and with well suited simulation might not vary at all.

The substantial variation as we mentioned before is usually three times standard deviation from the mean of the distribution. But when it comes to HTTP requests how do we decide the distribution of requests ? First of all we need to identify in which attributes do requests vary and then create enumeration of the observations of each attribute in order to create their distribution. We could also take combinations of attributes and enumerate these, where the most common combination of attributes would situate in the distribution around the mean and some unseen combinations of attributes could mean an anomaly. In our case we will use an autoencoder to determine reconstruction error of normal and anomalous data, where we will calculate the mean of normal data reconstruction errors and compare attack data reconstruction error with threshold, determined by adding multiples of standard deviation of the distribution of normal data reconstruction error's. We will discuss more about determining this threshold in the implementation chapter 5.

Addressing multi-attribute data problem is cut off due to the fact that we will at first only use HTTP request URIs for modeling normal system behavior, therefor only one

attribute is present. In case we would like to use multiple fields from HTTP request, we could still create sequences of multiple attributes, or combine multiple models which would process attributes on their own. Similar approach is used in [6] which we will examine later.

The biggest concern is raised when it comes to addressing changes that happen over time. It would be great if system, that model will be created for would never change, but in information technologies world, implementations change fast. This creates need for easy to update model representing what we consider as normal behavior. In practice it means, as we will use machine learning, retraining the model.

3.2 Anomaly Detection Outcomes

When system norm is defined and anomaly detection algorithm applied, there are three possible outcomes, that should be taken into consideration [18]:

- Correct detection
- False Positives
- False Negatives

Correct detection is a desired outcome, where detected abnormality in data is in fact anomalous and is not part of the expected process. In real-life systems it is impossible to achieve anomaly detection where only the correct detections occur. This leaves room for false positives and false negatives. False positive outcome in anomaly detection happens to look anomalous, but is accepted by our perception of the intentional behavior of the system. On the other side of unwanted outcomes are false negatives. In this case anomaly detection doesn't catch an anomaly that occurred in the system, due to abnormality being insufficiently significant [18].

To address false positives and false negatives we can aim to only estimate the possibility of sample being anomalous rather than simply answering whether it is abnormal or not.

3.3 Anomaly Detection Approaches

In this section we will take a look at anomaly detection approaches in computer networking systems and cyber-security as it is important to review all the possibilities and characteristics, that have been researched, before designing own solution.

According to [18]The main approaches of anomaly detection can be divided into three primary groups:

- Distance-based
- Density-based
- Rank-based

Distance-based group of approaches suppose that points that are farther from others are considered more anomalous. This Approach addresses fuzziness of anomalies, and take into account that some anomalies might be more or less anomalous as others. In density-based group, points are to be more anomalous when they lie in relatively low density regions. Rank-based approaches state, that the nearest neighbours of the most anomalous points have different nearest neighbours, than these anomalous points [18].

The nature of the data for this approaches may vary in terms of supervision. Three cases of this data are following [18]:

- Supervised: training data possess classification labels and the comparisons and distances are with respect to labeled training data.
- Unsupervised: there are no labels known, therefor comparisons and distances are applied to entire data set.
- Semi-supervised: in this case there are provided some labels for example samples of new malware or malicious http request containing SQL injection and a semi-supervised learning algorithm may find similarities in other unlabeled cases and determine their membership in the same category.

In our anomaly detection attempt the data for learning the norm of HTTP traffic on server are of unsupervised character. The data are retrieved from penetration testing session of GREYCORTEX s.r.o. which was divided into simulating normal behavior of clients using web-system and attacking part. There are no provided labels, however for the training data we will only use the normal behavior part. Even though in unsupervised anomaly detection algorithm one of the characteristics that should be met is dynamically defined normal behaviors, in our work we should first experiment with the possibilities of implementing anomaly detection algorithm for given purpose and only then move to automatising the learning process.

For HTTP traffic detection it is meaningful to use distance-based anomaly detection approach. It is in place to ask a question how to determine whether one observed HTTP request is farther from another. There are multiple attributes, that could be taken into consideration in answering this question. How to decide the metrics for comparison of HTTP requests ? These questions are some of the critical for implementation of our anomaly detection algorithm.

3.4 Autoencoders

During the research of possibilities of anomaly detection in HTTP traffic one option stood out with its elegance and prospect of utilising deep learning techniques. It is apparently usage of autoencoders. Let's take a look at what autoencoders are, what different types of them are known and how can we use them in anomaly detection. The information we will use in this section is from [15].

An autoencoder is a neural network trained to attempt to produce output from its input. The word attempt is important because rather than autoencoder's output being

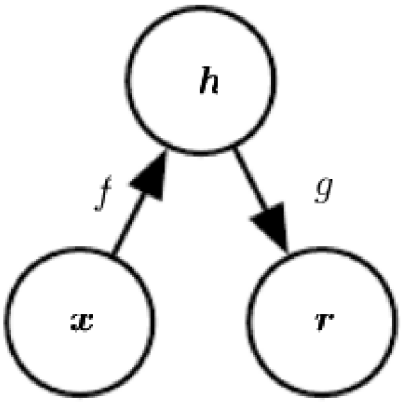

Figure 3.1: The general architecture of autoencoders. Encoding an input **x** with encoder **f** to internal representation or code **h**, decoding it with decoder **g** and mapping it to the output **r**. Adopted from [15]

identical copy of the same autoencoder's input, it can vary depending on reconstruction error. It wouldn't be especially useful to just copy inputs to outputs, therefor autoencoders are usually designed and trained to be unable to learn to copy perfectly. The models are accounting this fact forced to prioritize which aspects are important for the characteristic input and thus often learn the most remarkable and characterizing properties of the data [15].

Autoencoder has an internal hidden layer **h** that describes a code used to represent initial input. We can imagine this as some kind of compression algorithm. This neural network consists of two parts encoder and decoder. Encoder is the part already mentioned, which creates an internal representation of the input and decoder then produces reconstruction of the input providing it at the output of whole network [15]. This architecture can be seen in figure 3.1.

The main reason to construct autoencoders is apparently not copying inputs to outputs, but as we have previously mentioned, it may help in obtaining useful features from the data. One way to obtain such features is to constrain the dimension of internal representation of the input, making it smaller than an initial dimension of the input. Such an autoencoder is then called **undercomplete** and this type of an autoencoder is forced to capture the most relevant features of the training data. The Learning process of this type of autoencoder is then described as minimizing a loss function, where loss function is penalizing dissimilarities of the output of the decoder portion from input of the autoencoder. Loss function can for example be mean squared error. However if these autoencoders are given too much capacity, they struggle to obtain or learn any useful information from the input data. On the other hand if dimension of input data is equal to dimension of the hidden code, the autoencoder can learn to copy the input to output without learning anything useful about the input data. The same problem persists in **overcomplete** autoencoders, where the hidden code has dimension greater than input [15].

3.5 TensorFlow

In our work we will be using TensorFlow interface for implementing and running machine learning algorithms. It is important to introduce some of its features and terminology, as we will be using it in the design of our model and its data preprocessing in chapter 4, and the implementation chapter 5. Some of the terminology and theory applies to machine learning in general, but something is exclusive for TensorFlow. The majority of the information in this section comes from [21] and from TensorFlow online guides and blogs.

TensorFlow is a scalable and multi-platform programming interface, developed by the researchers and engineers of the Google Brain team, used for the implementation and running of the machine learning algorithms, including convenience wrappers for deep learning [21]. The *Tensor* in its name stands for multi-dimensional array, which has an uniform type, called *dtype*, where all tensors are immutable, meaning that you cannot update the contents of a tensor, only create a new one [25].

TensorFlow utilises high-level application interface (API) called **Keras**, which we will be using in achieving our goals in implementation. The initial release of Keras was as a standalone API, that could leverage Theano as a back-end, and later the support for TensorFlow was added. Theano is Python library used for defining, optimizing, and efficient evaluation of mathematical expressions involving multi-dimensional arrays [17].

Model in machine learning is a function with learnable parameters. It maps an input to an output. The training of the model on data, can then obtain the optimal parameters. In TensorFlow, one of the ways for creating a machine learning model, is by using the *Layers* API. The most common type of model is the **Sequential** model. It is a linear stack of layers, where each layer has exactly one input and output tensor. It is not appropriate when we need model with multiple outputs and inputs, or any of the layers within the model does [26].

3.5.1 TensorFlow Keras Layers

There are many different layers for use in TensorFlow models. In our anomaly detection solution and autoencoder neural network we will use four different layers. The Text Vectorization Layer, the Embedding Layer, the Flatten Layer for the text preprocessing and the Dense Layer for the construction of an autoencoder.

The Text Vectorization Layer provides basic options for managing text data in a Keras model. It is used to transform a batch of strings, where one sample of the data represents one string, into either list of token indices, or a dense representation. In list of token indices, each word of the string has its unique integer value assigned and in dense representation each sample is one dimensional tensor of float values representing data about the tokens. This layer provides ability to use its method *adapt()* on a dataset, analyzing the dataset, determining the frequency of individual string values and creating *vocabulary* from them. The vocabulary can have either limited or unlimited size. When the limit of the vocabulary is present, the layer will use the most frequent terms to create the vocabulary. This layer also contains method for text standardization which by default is lowering the case and stripping the punctuation. This layer is also capable of creating *n-grams*, which are slices of the string of the length **n** characters. In some modes, such as *binary*, *count*

and *tf-idf*, the layer by default pads the output to maximal number of tokens[27]. The Text Vectorization Layer of TensorFlow's Keras API is a little bit different from what we define as our text vectorization layer in chapter 4, but it is a significant part of it.

The Embedding Layer provides embedding functionality within Keras API. The word indices created by text vectorization layer, can be converted into input features in different ways. One of them is applying one-hot encoding to the token indices, converting them into vectors consisting of ones and zeros. If we are working with larger vocabulary, that contains many words, this vector will be of the size of this vocabulary, resulting in very sparse features. This is highly inefficient as all the features would be zeros, except for the one representing the word. The embedding is on the other hand much more effective, as we can map each word to a vector of fixed size. The size of the embedding vectors can be much smaller than the number of unique words within the vocabulary, to be able to represent them as input features. Since the embedding layer in a neural network is trainable, it can manage to extract salient features from these words and sequences [21].

The Dense Layer is a building block for our autoencoder architecture described in chapter 4. The TensorFlow guide [24] describes the dense layer as regular densely-connected neural network layer. A dense connection of layers means, that each neuron in a layer receives an input from all previous neurons. The dense layer applies *activation* function to the input and provides it to the output. The scheme of such activation function is following:

```
activation(dot(input, kernel) + bias)
```

where *input* is an input provided to a layer, *kernel* is a weights matrix created by the layer, *bias* represents a bias vector created by the layer and a *dot()* function calculates a *dot* product between the input and the weights [24].

Chapter 4

Design and Data

In this chapter we will introduce the data used for modeling the norm of HTTP traffic of targeted web server and the attack data used for testing anomaly detection outcomes. We will review the data retrieval and its following preprocessing needed in order to use it in the neural network. For this purpose we need to describe additional preprocessing layers for text vectorization and embedding and then we can introduce overall design of autoencoder neural network used for anomaly detection.

4.1 Data Source and Preparation

The data we use in our anomaly detection attempt comes from penetration testing session of GREYCORTEX s.r.o. provided by Petr Chmelář. The goal of this session was to create data suitable for optimization and testing of methods used for HTTP traffic analysis with emphasis on repeatability and possibility of automatized generation of normal as well as malicious HTTP traffic. Among the main goals was to obtain information about normal traffic, detection of security risks and vulnerabilities introduced and explained in [14] and chapter 2. For this purpose the team prepared safe environment including web server with representative applications, disconnected from the internet and automatisation and testing frameworks. The server(Apache) contained installed applications of content management system WordPress, e-learning system Moodle, Prestashop e-shop and intentionally vulnerable application WebGoat.

In the first part of the session divided into repeatable experiments focused on generating unharmful HTTP traffic by using various browsers including Firefox, Chrome, Internet Explorer and Safari. There was around twenty scenarios committed on all applications with seventy-eight executions. For example in case of WordPress creation of page, adding or removing items in the e-shop, purchasing or canceling the order before payment, etc. In the second part of the penetration test, separated by time gap, were tested vulnerabilities of web applications using open and free to use tools for penetration testing and vulnerability detection such as OpenVAS, Nessus, Metasploit, Hail Mary, Hydra, SQLmap, BurpSuite and more.

Besides the side scripts gathered using Selenium framework allowing for further generation of more data, the team obtained packet capture files of the network traffic during testing. This packet capture files are source of the data used for modeling the normal

behaviour of HTTP traffic in our neural network. These were then parsed into Comma Separated Values(CSV) format using network protocol analyzer **TShark**. Using this tool we separated the HTTP layer of the packet. Initially the intentions were to use multiple HTTP request header fields, but after consultation with Petr Chmelář, we decided to only use HTTP request URI field in the beginning of such project. We agreed, that in case of successful modeling of HTTP traffic norm and anomaly detection based on created model using request URIs, we can design and implement more complex anomaly detection algorithm with other request header fields in the future, detecting other different types of anomalies in HTTP traffic. The packet capture files were fortunately divided into parts where normal traffic was separated from the one with attacks, so we could easily parse these files one by one with TShark into separated CSV files with following command line:

```
tshark -r 1.pcapng -Y 'http.request.method == POST or http.request.
    ↪ method == GET' -T fields -e frame.time -e http.request.uri -E
    ↪ header=y > normal.csv
```

Where **-r** option specifies a capture file from which to read. **-Y** is a display filter which selects packets matching following filter. In our case we choose to use HTTP requests whose request methods are either POST or GET. **-T** option is used to set the format of the output of decoded packet data, in form of **fields**, followed by options **-e** specifying the values to choose, that would create columns of these values [7].

Now as we know the steps needed for preparation of our data for their usage in our anomaly detection, we can move to describing the design of the data preprocessing.

4.2 Data Preprocessing

The nature of our data is in text form. With previously mentioned preparation, we obtained data set in comma separated values format, consisting of strings representing chosen request header fields. Then each header field has its own place and in case of construction of a data set will represent one column. In order for neural network model in form of an autoencoder to be able to work with the data provided, the data cannot be in form of a string. First of all we need to be able to represent words within the strings as numbers and then create a vector out of them, that can be passed into the model input.

The processing of samples, which in our case are HTTP request URIs [1], contains the following steps:

- Standardization of samples.
- Splitting samples into substrings (words).
- Tokenization of substrings, which also includes indexing of the tokens in form of associating unique int values with tokens.
- Transformation of tokenized samples using this index into dense float vector.

[1] We do not use full request URIs including protocol specification and host as this information does not change throughout the whole data. Our request URIs are all from the same server.

These steps are then processed within additional preprocessing layers that add to the core autoencoder neural network. The main two layers are The Text Vectorization Layer and The Embedding Layer. Our preprocessing design also includes the layer used for flattening the output of the embedding layer, but since it only serves as dimension reduction, we will describe it alongside with the embedding layer.

4.2.1 The Text Vectorization Layer

The text vectorization layer of TensorFlow is responsible for standardization of samples with standardization method and tokenization of substrings. In our design we can differentiate between the text vectorization layer of our architecture and the text vectorization layer implemented in TensorFlow. The layer from TensorFlow needs to be provided with samples divided into words to be able to tokenize them. First we need to decide how to split HTTP request URI string into such words. Then the layer will be able to create vocabulary from this words. However the text vectorization layer of our architecture, is defined as all the steps of the preprocessing, before training TensorFlow's text vectorization layer and the final vectorization using this layer.

In natural language text preprocessing the substrings that would samples be split into would represent words of the sentences, which would represent the samples. We do not process natural language and instead of sentences we posses HTTP request URIs. They have parts separated with numerous special characters, in order for interpreters being able to distinguish these parts. We can also use this parts as our substrings and define separator as one of the listed special characters:

```
"\, /, +, =, ?, &, %, . and ,"
```

However we cannot split with:

```
" - " or " _ "
```

as these characters are included within some of the names in our data set, therefor these would be split incorrectly.

Following URI sample:

```
/moodle/mod/quiz/processattempt.php?cmid=30796
```

will then transform into sequence of words, separated by white space character:

```
moodle mod quiz processattempt php cmid 30796
```

The standardization method of natural language text input usually includes lower-casing and punctuation stripping. We do not have reasons to do otherwise, therefor standardization part of our Text Vectorization Layer transforms samples into lower case and strips the punctuation.

Large data sets usually need to have vocabulary size specified as there would be too many words. Then the most common words are chosen and according to their count within given data set, assigned a token. In our case we are not operating with very large data set

and can use all words, that we managed to get from splitting URI samples. This would leave us with vocabulary of size 8252 words.

After passing our input data into the layer, in form of sequences of the words, we will get sequences of integer indices. Now these sequences are of variable length. However, we need to have sequences of the same length for our neural network. The last step of this part of the preprocessing of the data, would be to use padding to compensate for shorter sequences with zeros, attaching them to the end of each incomplete sequence, to the length of the longest sequence within the data set.

4.2.2 The Embedding Layer

The Embedding layer of our model is responsible for transformation of tokenized samples using positive integers (indices) assigned to them, into dense vectors of fixed size.

Some of the parameters when designing such a layer are input dimension, output dimension and whether or not are masking and padding used for the input data of the layer. By crafting the text vectorization layer we also received its vocabulary as the part of it. The length of resulting vocabulary will then represent the input dimension of our embedding layer.

The next task is deciding the size of the output dimension, which would represent number of floats within each vector, representing the word in the sequence of the sample. This value is usually empirically decided and adjusted for the purposes of the designed model and according to its results. However [23] proposes „general rule of thumb“ about the number of embedding dimensions. They state, that the embedding vector dimension should be the fourth root of the number of categories. The number of categories is in our case the number of words in our vocabulary. So when we calculate the following formula:

$$\sqrt[4]{V} = \sqrt[4]{8252} \cong 9.5310 \approx 10$$

where **V** is the length of our vocabulary, we will get approximate dimension size for our embedding layer. We can then experiment during the implementation by either increasing or decreasing this number and see, what works best for this concrete problem.

The output of the embedding layer would then lead to the input of the flattening layer, which reduces the dimension of the data from shape *(samples, words, vectors)* to *(samples, words*vectors)*.

4.3 The Autoencoder Architecture

After the data is retrieved from CSV file, preprocessed with The Text Vectorization Layer and The Embedding Layer it can be used for training the autoencoder. It is important to say, we need to train the autoencoder only with the normal, unharmful HTTP traffic data, as the principle of our anomaly detection is evaluation of the reconstruction error produced by the autoencoder. The hypothesis is, that in attempting to reconstruct the attack data

with its internal representation of the normal data, the reconstruction error will be higher by at least set threshold, than the reconstruction error of the normal data reconstruction.

The autoencoder neural network, shown in the figure 4.1, is divided into two *Sequential* models, an *encoder* and a *decoder*. The encoder is responsible for dimension reduction of the data provided to its input. By doing this, it can learn some useful properties of the data. It consists of four *Dense* layers, where one of them is an input layer and remaining three are hidden layers. Each one of these layers cuts the previous dimensions of the input into half. The output of the last hidden layer of the encoder, is connected to the input of the decoder.

The decoder is responsible for attempting the data reconstruction. It as well as the encoder consists of four layers. The autoencoder parts can be used separately, but in the mode of autoencoder, the input layer of the decoder is hidden. This input layer is then followed by two additional hidden layers, where the dimension of their inputs increase two times over each layer. The last, output layer of the decoder, has the same number of output neurons as the number of features provided on the input of the autoencoder. Every layer of the autoencoder uses activation function *relu*, except for the output layer, that uses *linear* activation function.

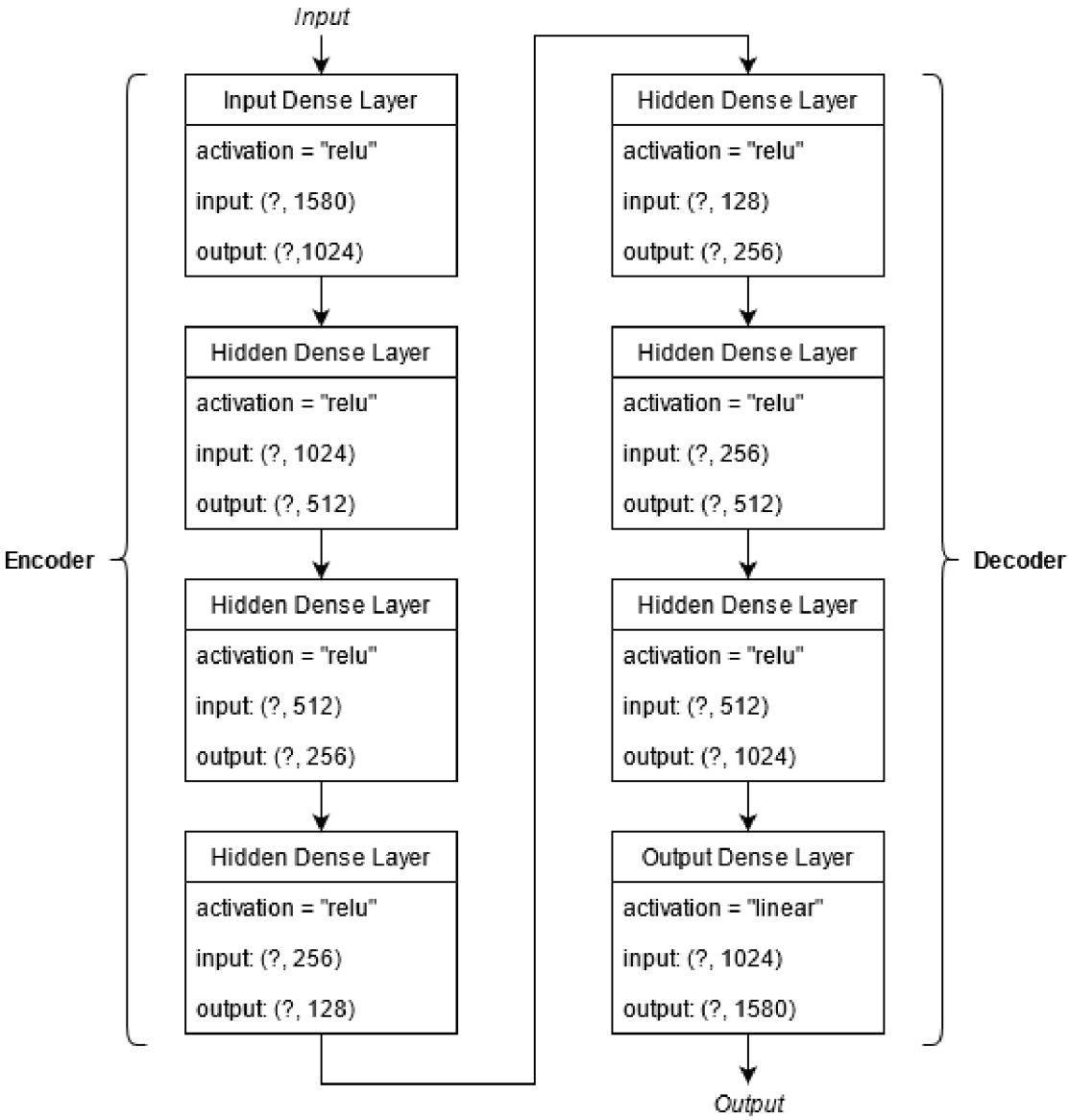

Figure 4.1: The Autoencoder Neural Network's architecture. Each list in the graph represents one layer of the autoencoder. The additional information provided in each layer is activation function and input and output shapes. **Left**: *encoder* part, **right**:*decoder* part of the model.

Chapter 5

Implementation and Results

This chapter contains information about implementation details and the results of anomaly detection algorithm. We will introduce some of the remaining tools, used to accomplish the implementation goals, that have not yet been mentioned. We will also discuss some of the workarounds needed in the implementation and possible improvements for the future.

5.1 Implementation Tools

The implementation is realised in programming language **Python** as it is one of the most popular programming languages for data science. Even though the performance of interpreted languages, which Python is, for difficult computation tasks is lower than the performance of lower-level programming, its extension libraries such as NumPy, built on lower layer Fortran and C implementations, are enabling fast, vectorized operations on multidimensional arrays. The version of Python used in our implementation is **Python 3.8**.

The data extraction from packet capture files provided by GREYCORTEX s.r.o. is achieved by using **TShark**, network protocol analyzer. This tool is listed here for the completion of tools used, but it was not used in the implementation part. It was used in order to prepare suitable data set for the model and its use is described in chapter 4.

The data extraction from CSV file, its representation in form of dataframes and operations on this data is provided by open source Python Data analysis Library, **Pandas**. For the data manipulation, in form of high-level mathematical functions, is used Python library **NumPy**. The version number of NumPy used during the implementation is **1.19.4**. Machine learning library **Sklearn** provides our implementation with function for splitting the training and the test data. For graphical outputs is used Python plotting library **Matplotlib**.

5.2 Implemented Functions

Our anomaly detection is implemented in Python script *anomalydetection.py*, which consists of multiple functions needed for the realisation of the data preprocessing, autoencoder training and evaluation of the test and attack data. These functions are following:

- `get_data(filename, encoding)`
- `df_to_dataset(dataframe, shuffle, batch_size)`
- `vectorize_text(dataset)`
- `prediction(model, normal_data, attack_data, test_data)`
- `fit_model(model, epochs, batch_size, data)`
- `plot_loss(history)`

The `get_data()` function requires one positional argument *filename*, specifying the location of the data in CSV format. It reads this file and parses it into dataframe using pandas function `read_csv()`.It has one more optional argument which, specifies the encoding of the data to be used (default encoding is *UTF-7*). The dataframe has three initial columns, as they are present in the dataset. The columns are HTTP request method, HTTP content-length and HTTP request URI. We agreed to use only HTTP request URI in our implementation as usage of the other two, would create more complex implementation requirements. Therefor, this function drops this columns and also removes rows of the dataframe, that contain *None* values. Part of the preprocessing from the Text Vectorization layer, that we have defined in chapter 4, already happens in this function. This function also iterates over the HTTP URI strings within dataframe and splits them into sequences of words separated by white space character. The function returns retrieved and processed *dataframe*.

Another function in the list is the `df_to_dataset()` function. It is the function used to transform pandas *dataframe* into TensorFlow *dataset*. This transformation is needed for the data used for creation of vocabulary, as it is the easiest way, how to provide the sequences of split words, into `adapt()` method of TensorFlow's *TextVectorization*. The function requires one positional argument, *dataframe*, which specifies the dataframe to be transformed. The function also provides ability to shuffle data within the dataset. To enable shuffling, the optional boolean argument *shuffle* needs to be set to *True*. The function also provides combining consecutive elements into batches and pre-fetching of the elements, which helps to improve latency as later elements of the dataset are prepared at the time of processing of the current element. The last optional argument specifies the batch size. Throughout other parts of the implementation, except from the adapting of the text vectorization layer, the implementation uses pandas dataframes. However, this function might come in use, in case of further implementation iterations.

The `vectorize_text()` function is responsible for initialization of the text vectorization layer. It either uses `adapt()` method of *TextVectorization* or directly passes the vocabulary into initialization of the text vectorization layer. The location of vocabulary to be used, is specified by the optional argument *vocabulary*. It is also capable of saving the newly created vocabulary, when the optional boolean argument *save* is set to *True*. The saved vocabulary is in CSV format and at this point for read-only purposes. In order for this saved vocabulary to be used, slight adjustments to the functions are required. The function returns initialized text vectorization layer, ready to be used.

The function essential for the anomaly detection is `prediction()`. It is used for the evaluation of the attempts to reconstruct the attack data and the test data by the autoencoder. It requires four positional arguments. The first argument, *model*, specifies the

autoencoder used for the prediction. The remaining three arguments are for the normal, attack and test data, in this order. First of all the function uses the autoencoder to predict the normal data. For this the method `predict()` of the autoencoder is called with the input set to the normal data. This method returns reconstructions predicted by the model, which should be very similar to the input data. Then the function for calculation of the mean squared error, `mse()`, between the input normal data and the predicted reconstructions of the normal data is called. This function is included within Keras losses functions. This will leave us with the list of mean squared error values. The majority of this values, should be relatively low, as the autoencoder model, provided to the function, was trained on the normal data. The function then proceeds to calculation of the fixed threshold, which is the value, used to decide if the reconstruction errors of the upcoming test data, are high enough to be considered anomalous. In other words, if the reconstruction error between the input test data sample, is higher by this threshold than the reconstruction error of the predicted test data sample, this sample is considered anomalous. This value is calculated by adding the mean of the reconstruction errors of the normal data to one standard deviation of the reconstruction errors of the normal data. The final calculation is comparing the reconstruction errors of the samples of the test data and the attack data with the threshold. The function does not have return value, but it prints the percentage of the anomalies within the test data and the attack data on the standard output. It also plots the graphs of the distribution of the reconstruction errors of the attack and the normal data.

The `fit_model()` function is used to define callback function, in form of an early stop, that finishes the training of the model in case of unchanging validation loss, with patience of ten epochs. The next step within the function is to call `fit()` method on the model, which is part of Keras *Model* and is responsible for training the model on the input data. Our function requires four positional arguments. The arguments are *model*, *epochs*, *batch* and *data*. The model argument is expected to be compiled form of Keras model, as in this function it will be fit with the data and trained. The second argument specifies number of epochs of the training cycle. The third argument is used to define batch size of the data, in order for more optimal data handling. The last argument expects input data, that the model will be trained on. The function returns a history object, which is an attribute to record the training loss values and metrics values at each successful epoch. It also contains validation loss values and validation metrics values as we opt to monitor it.

The last function `plot_loss()` is used to plot the losses values during the training. It requires one positional argument *history*, which is returned by the `fit()` method of the model. It pictures the evolution of train loss and validation loss values, over the epochs, in single graph. This function does not have return value.

5.3 The Autoencoder

The implementation of the autoencoder, shown in listing 5.1, within our script is in form of python *Class* called *AutoEncoder*. Its base class is TensorFlow's *Model*. *AutoEncoder* consists of two attributes, in form of keras sequential models, the encoder and the decoder. It also defines method `call(self, x)`, which is responsible for autoencoder's logic. It receives input data x and encodes them using its encoder. Following is the decoding of internally encoded data and finally the function returns data as *decoded*. The output shapes

of the layers within the autoencoder can be seen as the first argument in `Dense()`. The second argument is the activation function for each layer.

```
class AutoEncoder(Model):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.encoder = tf.keras.Sequential([
            tf.keras.layers.Dense(1024, activation="relu"),
            tf.keras.layers.Dense(512, activation="relu"),
            tf.keras.layers.Dense(256, activation="relu"),
            tf.keras.layers.Dense(128, activation="relu")])

        self.decoder = tf.keras.Sequential([
            tf.keras.layers.Dense(256, activation="relu"),
            tf.keras.layers.Dense(512, activation="relu"),
            tf.keras.layers.Dense(1024, activation="relu"),
            tf.keras.layers.Dense(1580, activation="linear")
            ])

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Listing 5.1: Implementation of the autoencoder class.

5.4 The Script for Anomaly Detection

So far we have explained the implementation of the functions within the script, the autoencoder class and now, we have to take a look at its core. The algorithm part of our anomaly detection, is situated in main function of the **anomalydetection.py** and it uses functions explained in section 5.2, for its tasks. In this section we will explain behavior of this main function, which includes retrieving the data, preprocessing it, training the model with the normal part of the data and evaluating the predictions.

The first thing that the script does is, that it uses `get_dat()` function in order to retrieve HTTP traffic, that has been previously merged into one CSV file called *data.csv*. The initial part of the preprocessing already happens in this function. It prepares the sequences, that are in form of HTTP URI strings, by splitting them into words. This is closely explained in the chapter 4. This data is then split into two pandas *dataframes*, to separate the normal traffic from malicious. It was achieved by manually searching for the time gap within the packet capture files, as it is how GREYCORTEX s.r.o. team, that performed the penetration tests on the web server, decided to separate normal traffic and attacks on the server.

The portion of the data within the *dataframe*, that represents normal HTTP traffic is then split into training and test data, where the test is fifteen percents from total amount of normal data. Some of the functions within the script require to know maximal length

of the sequence, consisting of words within each sample, therefor the longest sequence was identified and *maxlen* variable set to *158*.

The script then proceeds to converting the training portion of normal data into dataset, using function `df_to_dataset()`, as it is needed for use by the text vectorization layer of TensorFlow. The next step is to adapt new text vectorization layer, which will be used for this model, either by using vocabulary locally stored, or by using `adapt()` method of the *TextVectorization*. All of this does function `vectorize_text()` which is called with our newly formed dataset. It returns layer which the script uses for text vectorization of each parts of the data separately.

After the text vectorization layer is successfuly created and trained with normal data, resulting in creation of *vocabulary*, the length of its vocabulary can be used to initialize the embedding layer of TensorFlow.

Following step of the script is, that on all the data parts, which are normal training data, normal test data and attack data, script applies preprocessing layers. First the text vectorization layer is used, followed by padding the sequences to *maxlen*, then the embedding layer is applied and finally, the `Flatten()` reduces the shape of the output from the embedding layer by one dimension.

In the next part an instance of *AutoEncoder* is created. This is then compiled into model, using *adam* optimizer and *mse* as a loss function. The model is then trained on normal train data, with recommended number of epochs to be two hundred [1]. Straight after the training of the model is finished, `plot_loss()` function plots validation loss and train loss progress during the training.

Up until this point, the script has preprocessed the input data, split it into different parts and created the model, that is trained to be able to encode and reconstruct the data, that is similar to our normal HTTP traffic sample. The last task of the script is to call the function, responsible for prediction, `prediction()`. This function proceeds to try three different encoding-decoding attempts on all of our three parts of the data. First the normal data reconstruction, from which can be calculated the threshold to detect the anomalies, as mentioned in section 5.2. The function and the script then prints the result of the anomaly detection to standard output in form of two lines, with percentage of detected anomalies within attack and test data. It also plots some kind of distribution, in form of histogram, of the reconstruction errors of normal and attack data.

5.4.1 Further Improvements

The implementation of the anomaly detection will be in future most likely split from the modeling of normal behavior. However, this implementation was only an initial experiment, to determine if such anomaly detection can be successful. The script for anomaly detection can be significantly improved for features such as automatic retrieval of the traffic and its

[1] The number of epochs is relatively high due to the fact, that we want to achieve high level of reconstruction of normal data and teach the model its most salient features. During the trial-error part of the implementation, experiments with lowering the number of epochs resulted in lower capability of the model to detect the anomalies.

evaluation. It could also be able to mark single samples of anomalies within live traffic. The modeling part then could be re-trained on larger portions of the data, and for example trained for an anomaly detection within whole web system implementation scenarios, that use different kind of applications. Also the current state of the script requires for every experiment conducted, the model to be retrained. This is not an issue if you are experimenting with the shape of the model and new data, as it would be required anyway.

5.5 The Results

The results of our anomaly detection experiment, that consisted of modeling the normal HTTP traffic of given web server, should reflect the capability of our detection mechanism, to differentiate between an anomalous and expected HTTP traffic. For this purpose we have split our training and test data, to be able to compare the prediction of percentage of anomalies, within test and attack data.

The portion of the test data is retrieved by a split function, which shuffles the samples within normal data, therefor we will at each iteration of re-training of the model, get slightly different approximations of the percentage of anomalies. This is also caused by the fact, that each time while using this function, we are training the model with different portions of the normal data, therefor the expected noise within test data, affects the resulting anomaly approximation differently. However, this helps us to verify, that the implementation of our autoencoder, gets its job done, in different layouts of the training data.

Let's take a look at different outputs of our anomaly detection, starting with the simple text output. When we try to run our script multiple times, with same parameters including number of epochs, the proportion between test and normal data, and the proportion between validation and training data, described in previous parts of this chapter, we will get fairly similar outputs, shown in listing 5.2. We achieved these results with setting the number of epochs to three hundred, splitting the normal and test data by fifteen percent and validation and training data by ten percent. The number of up to five percent of approximated anomalies within the test data, in each of these outputs, is caused by the fact, that we have truly small data set for modeling the normal network, therefor some scenarios of the application usage or some noise could propagate into this approximation of the percentage of anomalies. On the other hand, the percentage of anomalies within attack data is significantly higher. It is not near one hundred percent, because it is almost impossible to provide the server with only attack requests. It also characterizes the intensity of the penetration test and the sensitivity of our anomaly detection, to abnormally looking requests. After consultation of these results with Petr Chmelář, we agreed, that this approximated percentage of anomalies within the attack part of the data set, could represent the amount of malicious HTTP requests.

The following are graphical outputs in figures 5.1 and 5.2, that will help us to demonstrate the distribution of reconstruction errors within normal data and attack data reconstruction attempts. For this experiment, we have adjusted the percentage of the test data, split from the normal data, to five percent, to be able to use larger portion of the normal data for the training. The number of the epochs remained the same.

```
1. Percentage of anomalies in attack data: 61.72
   Percentage of anomalies in test data: 3.11

2. Percentage of anomalies in attack data: 69.94
   Percentage of anomalies in test data: 5.20

3. Percentage of anomalies in attack data: 70.61
   Percentage of anomalies in test data: 4.28
```

Listing 5.2: Listing of three different outputs, from separate model trainings.

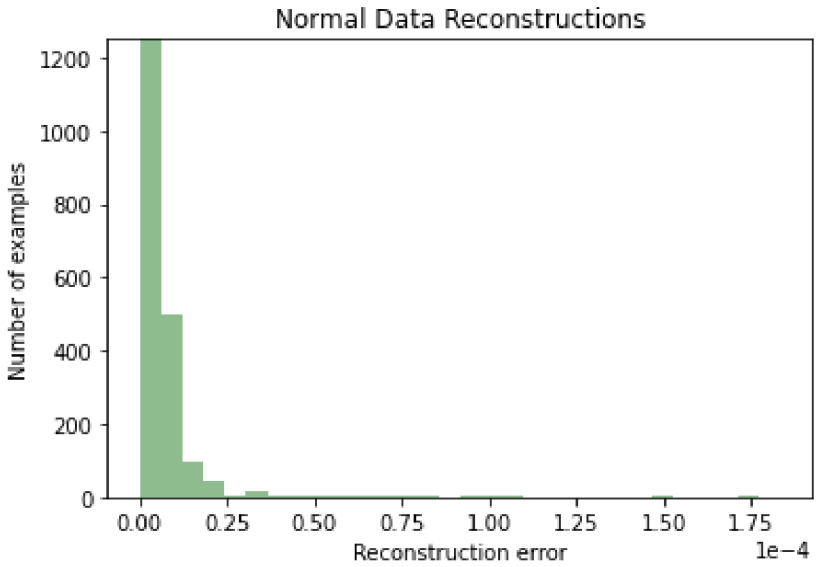

Figure 5.1: The distribution of the reconstruction errors of the autoencoder, for normal data samples. Note that reconstruction errors have metrics prefix **1e-4**.

The plots of the figures need to have limited number of examples to **1250**, for better visibility of higher reconstruction errors. As we can see in figure 5.1 the majority of the reconstruction errors lie within the range of **0** to **2.5e-5**. This is caused by the fact, that our autoencoder is trained on this data, therefor its reconstruction does not generate high error values. On the other hand the attack data reconstruction errors, shown in the figure 5.1, are more widely spread. The majority of reconstruction errors of the attack data samples is in range of **0** to **4e-4**. This is the result of high frequency of the anomalies within the data set and the autoencoder not being able to reconstruct such abnormal data.

The last picture, that can be seen in figure 5.3, shows the progress of the loss of the training loss and the validation loss functions during the training of the model. This information can indicate, whether or not we are under-fitting or over-fitting the model.

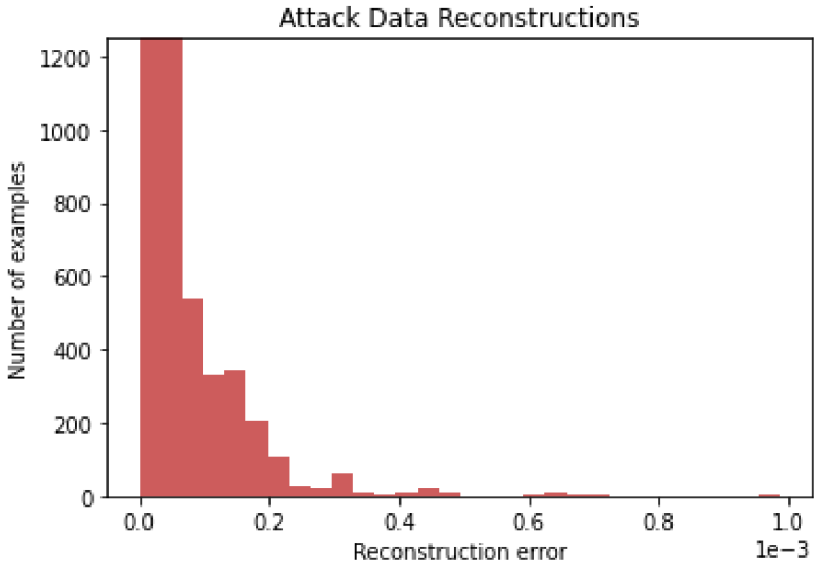

Figure 5.2: The distribution of the reconstruction errors of the autoencoder, for attack data samples. Note that reconstruction errors have metrics prefix **1e-3**.

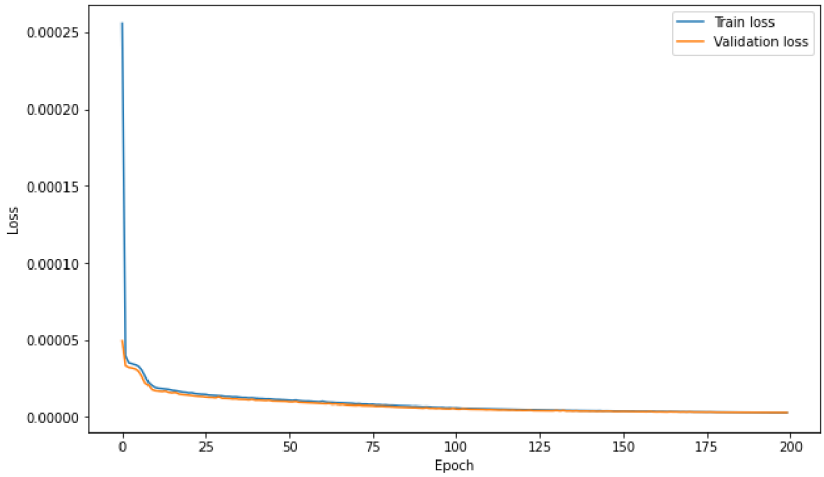

Figure 5.3: The graph of the training loss and validation loss functions progress, during the training of the model.

Chapter 6

Conclusion

Throughout this work we have reviewed the possibilities of an anomaly detection as one of the methods for intrusion detection in web systems. We have introduced autoencoder neural networks and explained, how they could be used for anomaly detection. What is more, we have managed to successfuly implement initial experiment, which consisted of modeling the normal behavior of HTTP traffic and been able to use it for anomaly detection of upcoming HTTP traffic.

However, there are many improvement possibilities that have yet to be done. First of all the implementation result in form of script, needs to be adjusted if it is to be used in real world. This would mean slight adjustments, where this script would be able to either routinely or dynamically retrieve HTTP traffic in order to attempt the reconstruction of its messages and detecting the malicious ones as anomalies. The modeling of normal behavior of HTTP traffic, should be separated from anomaly detection. The script would only be using already trained model for anomaly detection.

The model can be also used as a building block for more complex solution, including other different information from HTTP traffic. One of the limitation of our solution was relatively small data set. If we are be able to obtain larger data sets, we will have opportunity to build even more precise anomaly detection models, than what we have now. Hypothetically, we can try to create anomaly detection solution, based on autoencoder neural networks, that is independent on the web server implementation. On the other hand, we can try to create pre-trained models for concrete implementation scenarios, that would be deployed within particular web server implementations, with its specific applications.

Bibliography

[1] Alani, M. M. *Guide to OSI and TCP/IP Models*. 1st ed. Springer International Publishing, 2014. ISBN 978-3-319-05151-2.

[2] Barth, A. *HTTP State Management Mechanism* [Internet Requests for Comments]. RFC 6265. RFC Editor, April 2011. http://www.rfc-editor.org/rfc/rfc6265.txt. Available at: http://www.rfc-editor.org/rfc/rfc6265.txt.

[3] Belshe, M., Peon, R. and Thomson, M. *Hypertext Transfer Protocol Version 2 (HTTP/2)* [Internet Requests for Comments]. RFC 7540. RFC Editor, May 2015. http://www.rfc-editor.org/rfc/rfc7540.txt. Available at: http://www.rfc-editor.org/rfc/rfc7540.txt.

[4] Berners Lee, T., Fielding, R. T. and Masinter, L. *Uniform Resource Identifier (URI): Generic Syntax* [Internet Requests for Comments]. STD 66. RFC Editor, January 2005. http://www.rfc-editor.org/rfc/rfc3986.txt. Available at: http://www.rfc-editor.org/rfc/rfc3986.txt.

[5] Berners Lee, T., Fielding, R. T. and Nielsen, H. F. *Hypertext Transfer Protocol – HTTP/1.0* [Internet Requests for Comments]. RFC 1945. RFC Editor, May 1996. http://www.rfc-editor.org/rfc/rfc1945.txt. Available at: http://www.rfc-editor.org/rfc/rfc1945.txt.

[6] C., K. and G., V. *Anomaly Detection of Web-based Attacks* [online]. Reliable Software Group, 2003 [cit. 2021-3-10]. Available at: https://sites.cs.ucsb.edu/~vigna/publications/2003_kruegel_vigna_ccs03.pdf.

[7] Combs, G. *Tshark - Dump and analyze network traffic* [online]. 2019 [cit. 2021-05-09]. Available at: https://www.wireshark.org/docs/man-pages/tshark.html.

[8] Daren S. Starnes, D. Y. and Moore, D. S. *The Practice of Statistics*. 5th ed. W. H. Freeman and Company, 2014. ISBN 978-1-4641-0873-0.

[9] Fielding, R. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Authentication* [Internet Requests for Comments]. RFC 7235. RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7235.txt. Available at: http://www.rfc-editor.org/rfc/rfc7235.txt.

[10] Fielding, R. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests* [Internet Requests for Comments]. RFC 7232. RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7232.txt. Available at: http://www.rfc-editor.org/rfc/rfc7232.txt.

[11] Fielding, R. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing* [Internet Requests for Comments]. RFC 7230. RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7230.txt. Available at: http://www.rfc-editor.org/rfc/rfc7230.txt.

[12] Fielding, R. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [Internet Requests for Comments]. RFC 7231. RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7231.txt. Available at: http://www.rfc-editor.org/rfc/rfc7231.txt.

[13] Fielding, R. T., Gettys, J., Mogul, J. C., Nielsen, H. F., Masinter, L. et al. *Hypertext Transfer Protocol – HTTP/1.1* [Internet Requests for Comments]. RFC 2616. RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2616.txt. Available at: http://www.rfc-editor.org/rfc/rfc2616.txt.

[14] Foundation, O. *OWASP Top 10 - 2017*. Publication. OWASP Foundation, 2017. Available at: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf.

[15] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning (Adaptive Computation and Machine Learning series)*. 1st ed. MIT Press, 2016. http://www.deeplearningbook.org. ISBN 978-0262035613.

[16] Gupta, C., Singh, R. K. and Mohapatra, A. K. A survey and classification of XML based attacks on web applications. *Information Security Journal: A Global Perspective*. [online]. 29th ed. 2020, no. 4, [cit. 2021-5-8]. Available at: https://doi-org.ezproxy.lib.vutbr.cz/10.1080/19393555.2020.1740839.

[17] LISA, L. *Theano 1.0.5 - Project Description* [online]. 2020 [cit. 2021-05-09]. Available at: https://pypi.org/project/Theano/.

[18] MEHROTRA Kishan G, H. H. and SUBRAHMANIAN, V. *Anomaly Detection Principles and Algorithms*. 1st ed. Springer International Publishing, 2017. ISBN 978-3-319-67524-4.

[19] Postel, J. *User Datagram Protocol* [Internet Requests for Comments]. STD 6. RFC Editor, August 1980. http://www.rfc-editor.org/rfc/rfc768.txt. Available at: http://www.rfc-editor.org/rfc/rfc768.txt.

[20] Postel, J. *Transmission Control Protocol* [Internet Requests for Comments]. STD 7. RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt. Available at: http://www.rfc-editor.org/rfc/rfc793.txt.

[21] RASCHKA, S. and MIRJALILI, V. *Python machine learning : machine learning and deep learning with Python, scikit-learn, and TensorFlow*. 2nd ed. Packt Publishing Ltd., 2017. ISBN 978-1-78712-593-3.

[22] Rescorla, E. *HTTP Over TLS* [Internet Requests for Comments]. RFC 2818. RFC Editor, May 2000. http://www.rfc-editor.org/rfc/rfc2818.txt. Available at: http://www.rfc-editor.org/rfc/rfc2818.txt.

[23] TensorFlow, T. *Introducing TensorFlow Feature Columns* [online]. 2017 [cit. 2021-05-09]. Available at: https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html.

[24] TensorFlow, T. *Dense Layer* [online]. 2021 [cit. 2021-05-09]. Available at: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense?hl=fr.

[25] TensorFlow, T. *Introduction to Tensors* [online]. 2021 [cit. 2021-05-09]. Available at: https://www.tensorflow.org/guide/tensor.

[26] TensorFlow, T. *Models and layers* [online]. 2021 [cit. 2021-05-09]. Available at: https://www.tensorflow.org/js/guide/models_and_layers.

[27] TensorFlow, T. *Text Vectorization* [online]. 2021 [cit. 2021-05-09]. Available at: https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/TextVectorization.