Prevent XSS Attacks Using Machine Learning

Sigmund Vestergaard
Student No. 16140435
National College of Ireland

Abstract—In this paper we present a system using Machine Learning to prevent XSS attacks on web applications. A deep neural network is built using the Keras library in Python and training using the CSIC 2010 dataset and data retrieved from pastebin.org. This neural network is then used to classify incoming HTTP requests according to whether they are XSS attacks or not. It was found that the system we developed blocks most attacks. A test using 25 XSS vectors showed an 84% success rate, which can be improved upon. The improvement in our case would come from developing better Machine Learning classifiers.

Index Terms—Machine Learning, XSS, web, web application,.

1 Introduction

THIS article serves as documentation of the background for an application we have built, which uses Machine Learning to prevent XSS attacks on web applications.

Following on from the suggestions set out in Section F of (Nunan, Souto, dos Santos, and Feitosa, 2012). Further research has been conducted in (Khan, Abdullah, and Khan, 2017). Together these two papers will form the basis for an implementation of a software system that uses machine learning techniques to automatically classify and detect XSS attacks such that these can be prevented. A classifier will be trained using the CSIC 2010 HTTP (Giménez, Villegas, and Álvarez Marañón) dataset and various XSS vectors found on pastebin (Various), respectively. Then the efficiency of the classifier will be tested on a live system running a web application known to be vulnerable to XSS attacks.

We will start by placing XSS attacks in context and give an outline of what has been done so far in this area. Then we will described the architecture we have designed our prevention system around. After that we will discuss how we are using Machine Learning and the data sets we used to train the models we employ for classification of incoming HTTP requests. After that we will describe how we tested our application and give measures for how efficient it is in preventing XSS attacks. Finally we conclude what effect our intrusion prevention has had, what its strengths and weaknesses, respectively, are, and how it can be improved.

The system has been deployed at http://docker.math.fo/, where two vulnerable web applications have been deployed, http://docker.math.fo/dvwa/ and http://docker.math.fo/mutidillae. If you want to look at the code in order to deploy it yourself you can find it along with a `docker-compose.yml` file with necessary dependencies for deployment can be found on GitHub at https://github.com/sigmundv/xss-prevent-py. The web applications are running in docker containers, but our system has been deployed on the host server. See Appendix ?? for more detailed instruction on how to deploy the solution.

2 Background

Most of the text in this section has been adapted from (Khan, Abdullah, and Khan, 2017). Due to the advancement of modern computers and our dependence on internet based services, web applications have become a primary target for cyber criminals. According to (Symantec, 2016), about 430 million unique pieces of malware were discovered; a growth of 36% compared to 2014.

Cyber criminals use malicious code and malicious URLs to attack individuals and organisations for personal, financial and political gains. Detection of these attacks becomes an important security challenge in cybersecurity.

OWASP, Open Web Application Security Project, has ranked Cross-Site Scripting (XSS) as the third most critical vulnerability on their latest top ten list from 2013 – a new list is due in November 2017, but the top six vulnerabilities are not being amended from the 2013 list (OWASP.org, 2017a).

Currently XSS represents 43% of all reported vulnerabilities. XSS attacks work by injecting malicious scripts around benign code in a legitimate website in order to access cookies, session information and other sensitive information retained by the user's web browser. XSS attacks are applied to the client side of web applications whereas SQL injections – the most critical vulnerability on OWASP's top ten list – are server side attacks. XSS attacks are targeting the application layer of the network hierarchy seeking to exploit vulnerabilities in the application itself as opposed to the network connection itself.

About 70% of attacks are reported to occur at the application layer, and these attacks are executed by executing malicious JavaScript in a user's web browser, but another vehicle for XSS is malicious and obfuscated URLs (Nunan, Souto, dos Santos, and Feitosa, 2012). In contrast to different sorts of network based attacks, malicious JavaScript is difficult to detect (Schütt, Kloft, Bikadorov, and Rieck, 2012). JavaScript attacks are performed by examining the vulnerability and exploiting it using JavaScript obscuration techniques to evade detection, hence detecting such JavaScript in real-time is imperative. Current security solutions are fundamentally based on two different approaches, signature-based and heuristic-based detection.

The signature-based approach is based on the detection of unique string patterns within the attacker's code, hence this fails whenever a new unknown string appears in the code, which leaves the application vulnerable until the signatures in the detection system can be updated. Cybercriminals can exploit this gap in time where the application is vulnerable to launch many attacks. The heuristic detection approach is based on decision rules formulated by security experts.

The advantage is that this approach can not only detect currently know attacks, but also previously unknown attacks. On the other hand the disadvantages are that scanning and analysis of incoming traffic is slow and that it introduces a high number of false positives, which means that a piece of code is identified as malicious when it is not. Researchers have recently started using machine learning in the detection of malware

The advantage of using machine learning is that we can easily detect previously unknown malware based on what we have taught the machine learning model, i.e. classifier, about the nature of malicious JavaScript, hence it is a very useful approach for detection of malware of an increasingly polymorphic nature. Several approaches have been proposed for classification and detection of malicious JavaScript code such as (Rieck, Krueger, and Dewald, 2010), (Curtsinger, Livshits, Zorn, and Seifert, 2011), and (Fraiwan, Al-Salman, Khasawneh, and Conrad, 2012), but the problem with these approaches is the overhead in the time needed for detection.

Thus a new, more efficient approach is needed for classification and detection of malicious JavaScript in an XSS attack, and by following the method outlined in (Khan, Abdullah, and Khan, 2017) this project aims to build a new detection system that uses machine learning classifiers to detect malicious pieces of JavaScript injected into a web application.

The reason I settled on XSS is that it is a critical vulnerability focused on by OWASP and that there is not very much work in this area, so there is an opportunity to make something quite cutting-edge.

3 Technical Approach

3.1 Data Preparation and Training of Classifier

The claims made here about the efficiency of the selected approach are from the article (Khan, Abdullah, and Khan, 2017). The approach proposed here is light-weight in nature with minimal runtime overheads. Detection of XSS attacks is based on static analysis of a given JavaScript code from which features will be extracted and fed into the machine learning classifier that determines whether it is malicious piece of JavaScript code or not.

We are using the Keras library (Chollet et al., 2015) for Python to build our classifier. This is a machine learning library that builds on top of the TensorFlow library (Abadi, Agarwal, Barham, Brevdo, Chen, Citro, Corrado, Davis, Dean, Devin, Ghemawat, Goodfellow, Harp, Irving, Isard, Jia, Jozefowicz, Kaiser, Kudlur, Levenberg, Mané, Monga, Moore, Murray, Olah, Schuster, Shlens, Steiner, Sutskever, Talwar, Tucker, Vanhoucke, Vasudevan, Viégas, Vinyals, Warden, Wattenberg, Wicke, Yu, and Zheng, 2015) from Google. This is a library that can be used to build models based of deep neural networks. We use the more user friendly Keras library to build our model/classifier instead of using TensorFlow directly.

The data used to train the classifier comes from two different sources:

1) The HTTP Dataset CSIC 2010 (Giménez, Villegas, and Álvarez Marañón), which contains 36,000 normal requests and more than 25,000 anomalous requests including XSS attacks. We filtered out the XSS related requests from this dataset. See Listing 2 for an example of this dataset.
2) XSS vectors posted in various pastebins on pastebin.org (Various). See Listing 1 for an example of one of these datasets.

```
<iframe %00 src="&Tab;javascript:prompt(1)&Tab;"%00>
<svg><style>{font-family&colon;'<iframe/onload=
    ↪ confirm(1)>'
<input/onmouseover="javaSCRIPT&colon;confirm&lpar;1&
    ↪ rpar;"
<sVg><scRipt %00>alert&lpar;1&rpar; {Opera}
<img/src='%00' onerror=this.onerror=confirm(1)
<form><isindex formaction="javascript&colon;confirm
    ↪ (1)"
<img src='%00'&NewLine; onerror=alert(1)&NewLine;
<script/&Tab; src='https://dl.dropbox.com/u/13018058/
    ↪ js.js' /&Tab;></script>
<ScRipT 5-0*3+9/3=>prompt(1)</ScRipT giveanswerhere=?
```

Listing 1. Example of data from pastebin

```
GET http://localhost:8080/tienda1/publico/autenticar.
    ↪ jsp?modo=entrar&login=bob%40%3CSCRipt%3Ealert
    ↪ %28Paros%29%3C%2FscrIPT%3E.parosproxy.org&pwd
    ↪ =84m3ri156&remember=on&B1=Entrar HTTP/1.1
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.5;
    ↪ Linux) KHTML/3.5.8 (like Gecko)
Pragma: no-cache
Cache-control: no-cache
Accept: text/xml,application/xml,application/xhtml+
    ↪ xml,text/html;q=0.9,text/plain;q=0.8,image/png
    ↪ ,*/*;q=0.5
Accept-Encoding: x-gzip, x-deflate, gzip, deflate
Accept-Charset: utf-8, utf-8;q=0.5, *;q=0.5
Accept-Language: en
Host: localhost:8080
Cookie: JSESSIONID=AC0EEEDD09663CB36C67DD1B787B0CF5
Connection: close


POST http://localhost:8080/tienda1/publico/autenticar
    ↪ .jsp HTTP/1.1
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.5;
    ↪ Linux) KHTML/3.5.8 (like Gecko)
Pragma: no-cache
Cache-control: no-cache
Accept: text/xml,application/xml,application/xhtml+
    ↪ xml,text/html;q=0.9,text/plain;q=0.8,image/png
    ↪ ,*/*;q=0.5
Accept-Encoding: x-gzip, x-deflate, gzip, deflate
Accept-Charset: utf-8, utf-8;q=0.5, *;q=0.5
Accept-Language: en
Host: localhost:8080
Cookie: JSESSIONID=D6037A58019A61C4E5745B952029D61F
Content-Type: application/x-www-form-urlencoded
Connection: close
Content-Length: 118

modo=entrar&login=bob%40%3CSCRipt%3Ealert%28Paros
    ↪ %29%3C%2FscrIPT%3E.parosproxy.org&pwd=84
    ↪ m3ri156&remember=on&B1=Entrar
```

Listing 2. Example of data from the CSIC 2010 dataset

In order to use this data we need to build a matrix of features, i.e. transform the textual data into a set of numbers, because this is what the machine learning algorithm understands.

The feature matrix will be built by first picking out a series of strings that normally appear in XSS vectors. We call this the *vocabulary* and feed it into the `CountVectorizer` class of the scikit-learn (Pedregosa, Varoquaux, Gramfort, Michel, Thirion, Grisel, Blondel, Prettenhofer, Weiss, Dubourg, Vanderplas, Passos, Cournapeau, Brucher, Perrot, and Duchesnay, 2011) (Buitinck, Louppe, Blondel, Pedregosa, Mueller, Grisel, Niculae, Prettenhofer, Gramfort, Grobler, Layton, VanderPlas, Joly, Holt, and Varoquaux, 2013) library together with the datasets mentioned above. The `CountVectorizer` class will then go through each row of data and count the number of occurrences per row of each of the strings in the vocabulary. See Listing 3 for the vocabulary we are using. Each line in this vocabulary is what we referred to as the *features* earlier.

```
document.location
document.cookie
javascript:alert
&lt;script&gt;
&lt;/script&gt;
onpagehide
&lt;
&gt;
```

Listing 3. The vocabulary we use as basis for our classifier.

When we have the feature matrix we can feed that into the Keras library and train a neural network to predict whether a payload is an XSS vector or not. See Listing 4 for the Python code that is building a feature matrix given a dataset and a vocabulary, and Listing 5 for the Python code that is training the neural network.

```
def vectorize_data(data):
    """

    :param data: The data to vectorize; it should be
        ↪ a list of strings, one per line.
    :return: The CountVectorizer, which we need later
        ↪  in order to transform incoming requests
        ↪ to a feature vector
            The feature matrix based on the given
                ↪ vocabulary.

    """
    vocabulary = open(vocabulary_file, 'r').read().
        ↪ splitlines()

    vocabulary_lengths = list(map(len, vocabulary))
    min_length = np.min(vocabulary_lengths)
    max_length = np.max(vocabulary_lengths)

    count_vect = CountVectorizer(ngram_range=(
        ↪ min_length, max_length),
                                 analyzer='char',
                                 token_pattern=r'(?u).*\
                                     ↪ w.*\w+.',
                                 vocabulary=vocabulary)
    x_train_counts = count_vect.fit_transform(data).
        ↪ toarray()

    return count_vect, x_train_counts
```

Listing 4. Python function that builds the feature matrix based on the dataset.

```
def train_data(fname):
    """

    :param fname: Filename that we stored the feature
        ↪  matrix to.
    :return: The model trained by Keras.
    """
    dataset = np.loadtxt(fname, delimiter=',')
    nrows, ncols = dataset.shape
    X = dataset[:, :(ncols-1)]
    Y = dataset[:, (ncols-1)]
    model = Sequential()
    model.add(Dense(12, input_dim=(ncols-1),
        ↪ activation='relu'))
    model.add(Dense(ncols-1, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
        ↪ optimizer='adam', metrics=['accuracy'])
    model.fit(X, Y, validation_split=0.33, epochs=10,
        ↪  batch_size=10)
    return model
```

Listing 5. The Python function that builds a neural network and trains it using the feature matrix

Now we have the trained model and we are ready to incorporate it into our prevention system.

3.2 Intrusion Prevention System

In Section 3.1 we have described how we prepare a dataset and train a classifier for classification of HTTP requests. Now we will continue describing the architecture of the Intrusion Prevention System and how we use the classifier to block incoming XSS attacks.

The starting point of the system is to use a firewall to direct the incoming requests towards our application. For our purpose we use the `iptables` firewall on a Linux system. IPtables has a built-in target called `NFQUEUE`, netfilter queue, which has the purpose of forwarding incoming packets to a userland application that then processes these packets. In our case we look for packets with an HTTP layer, extract the Path (in the case of a `GET` request) or the payload (in the case of a `POST` or `PUT` request) and either accept the packet, or drop it, according to the verdict given by our classifier from Section 3.1.

To analyse the incoming packets we use a Python library called Scapy (Biondi, 2002). By default Scapy does not provide a straightforward way to extract the HTTP layer from packets, so for that purpose we use the library scapy-http (Invernizzi, 2014), which adds HTTP support to Scapy. This allows us to get the information described above from the packets, which we then pass to the neural network classifier.

Furthermore, in the case where the classifier is determining a packet to contain an XSS attack we store the payload used in a CouchDB database for later analysis. CouchDB stores the data in the form of JSON documents, and the structure of the documents can be seen in Listing 6.

```
def store_xss_vector(self, source, path, payload,
    ↪ category):
    """
```

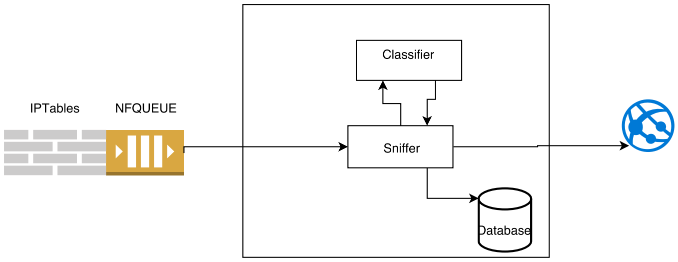

Fig. 1. Diagram showing the components of the Intrusion Prevention System and how they are interconnected.

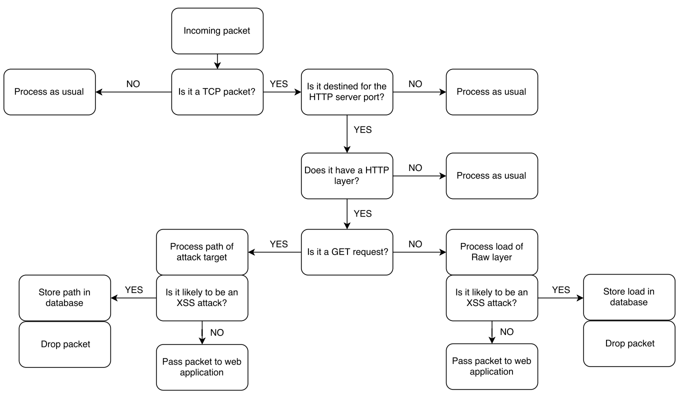

Fig. 2. Flowchart showing the flow of a packet through the Intrusion Prevention System.

```
    :param source: The IP where the attack originated from.
    :param path: The path of the page where the attack was performed.
    :param payload: The payload that was used in the attack, e.g. a pice of JavaScript.
    :param category: The category determined by the classifier; will always be one.
    """
    doc_id = uuid.uuid4().hex
    self.database[doc_id] = {'timeid': arrow.now().timestamp,
                             'source': source, 'path': path,
                             'payload': payload, 'xss_vector': category}
```

Listing 6. The Python function that stores information about XSS attacks in CouchDB.

Now when we have seen a description of the architecture it is helpful to illustrate the same in a diagra. See Figure 1 for this diagram and Figure 2 for a flowchart showing the flow in the system. Additionally, see Figure 3 for a class diagram showing the classes present in the system.

To round off this section we can take a look at the Python functions that do the bulk of the work, the `analyze_packet` and `classify_packet` functions, respectively. The former (see Listing 7) strips out the payload from the incoming packet and returns it to the latter function (see Listing 8), which passes it to the classifier, and then accepts or drops the packet depending on the verdict returned by the classifier.

```
def analyze_packet(self, packet):
    """
```

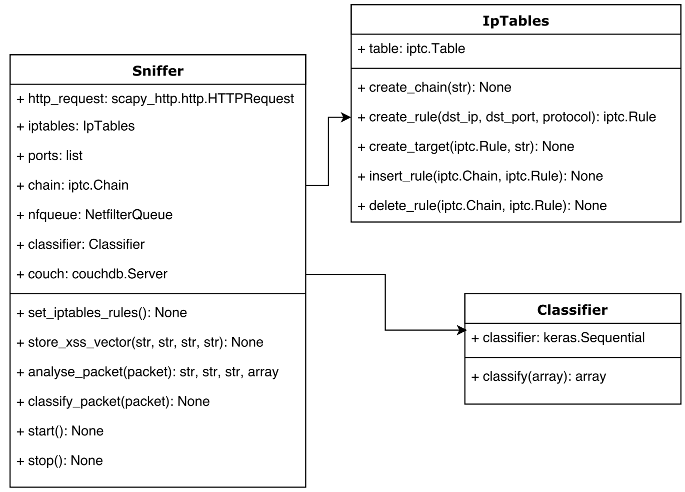

Fig. 3. Class diagram showing the classes being used in our application..

```
    :param packet:
    :return:
    """
    payload = packet.get_payload()
    pkt = scapy.all.IP(payload)
    source = pkt.src
    if pkt.haslayer(self.http_request):

        http_request = pkt.getlayer(self.http_request)
        host = http_request.Host
        method = http_request.Method
        path = http_request.Path

        if method == b"GET":
            path_split = path.split(b"?")
            path = path_split[0]
            load = b"?".join(path_split[1:])
            logging.debug("%s request path found: %s", method, path)
            return source, host, path.decode("utf-8"), [html.escape(unquote(unquote(load.decode("utf-8"))))]
        else:
            raw = http_request.lastlayer()
            try:
                load = raw.load
                logging.debug("%s request payload found: %s", method, load)
                return source, host, path.decode("utf-8"), [html.escape(unquote(unquote(load.decode("utf-8"))))]
            except AttributeError:
                logging.exception("No payload in raw packet")
                raise HTTPPacketException
    else:
        raise HTTPPacketException
```

Listing 7. The analyze_packet function, which strips out the payloyad from a packet.

```
def classify_packet(self, packet):
    """

    :param packet:
```

TABLE 1
A test of a sample of XSS vectors from the XSSer tool.

No. of XSS vectors tested	Percentage blocked by our system
25	$\frac{21}{25} \cdot 100\% = 84\%$

```
        :return:
        """
        try:
            source, host, path, payload = self.
                ↪ analyze_packet(packet)

            logging.debug("Host␣is:␣%s␣;␣payload␣is:␣%
                ↪ s", host, payload)
            category = self.classifier.classify(
                ↪ payload)
            if category:
                logging.info("XSS␣payload␣detected␣;␣
                    ↪ attack␣vector␣stored␣in␣DB␣and␣
                    ↪ packet␣dropped")
                self.store_xss_vector(source, path,
                    ↪ payload[0], str(category[0][0])
                    ↪ )
                packet.drop()
            else:
                logging.debug("Packet␣accepted")
                packet.accept()
        except HTTPPacketException:
            packet.accept()
```

Listing 8. The classify_packet function, which passes a packet payload to the classifier and drops or accept the packet based on the returned verdict.

4 Evaluation

Here I have decided to do a visual evaluation showing the injection of some XSS vectors into a text field in the vulnerable web application *mutillidae*. We'll be using the path mutillidae/index.php?page=dns-lookup.php to test on. The video showing this visual testing can be found on GitHub together with the code in a subdirectory called `evaluation`. The link can be found in the Introduction, Section 1.

You will find two videos in the mentioned subdirectory, one showing the testing of reflected XSS attacks and one showing the testing of persistent XSS attacks. The filenames will indicate which is which. For the small sample of attacks we test in the video our system blocked all of them.

We planned to do automatic testing using the tool XSSer (psy, 2010) in order to evaluate the success rate of our Intrusion Prevention System, but there were issues getting XSSer to evaluate the performance correctly, so we have ended up taking a small sample of the XSS vectors that XSSer is using and typing them in manually to see the response.

The results of these manual tests are summarised in Table 1.

In order to give a more comprehensive analysis of the performance of our system we would have to insert a lot more XSS vectors – the XSSer tool tests around 550 vectors, it is not feasible to do this manually.

5 Conclusion

After having gone through the background for our work, the design of the system, and some results of evaluating it, we are ready to talk about some lessons learnt during the work and also to point forward to further work that can be done in this area.

5.1 Lessons learnt

Since the work on this project was started the OWASP Top 10 for 2017 (OWASP.org, 2017b) has been released, and on this list XSS has been relegated for second place in 2013 to seventh place today. From this we can say that the importance of this work has diminished a bit since it was undertaken, since OWASP have prioritised it lower on their list of the top web application vulnerabilities.

During the work it was also discovered that analysis of SSL packets is challenging, and neigh on impossible if we are dealing with a security conscious user that has configured their web server correctly to only use ECDH or ECDHE ciphers we don't have a chance to decrypt the traffic even though we would have access to the private key since our solution is deployed serverside. An alternative could be to redirect the traffic through a proxy on the server and get the decrypted traffic that way, but we haven't looked into that, hence our solution is currently limited to servers without SSL configured, which makes its usability limited to the security conscious user, who would naturally have SSL enabled on their web server.

5.2 Further work

Three things could, and should, be done to develop the current solution further:

1) First upon dropping a packet we should give proper feedback to the client instead of just letting it time out.
2) The Machine Learning classifier should be improved by adding more features to it, i.e. the vocabulary should be expanded with more keywords from XSS attacks. In addition to that more Machine Learning algorithms should be trained and compared to the current one.
3) The solution should be expanded to also deal with SSL traffic, which is de-facto for websites in 2017.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Phillipe Biondi. Scapy, 2002. URL http://www.secdev.org/projects/scapy/. Accessed: 2017-12-12.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

François Chollet et al. Keras, 2015. URL https://github.com/fchollet/keras. Accessed: 2017-12-12.

C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX Conference on Security (SEC '11)*, 2011.

M. Fraiwan, R. Al-Salman, N. Khasawneh, and S. Conrad. Analysis and identification of malicious javascript code. *Information Security Journal*, 21(1):1–11, 2012.

Carmen Torrano Giménez, Alejandro Pérez Villegas, and Gonzalo Álvarez Marañón. Http dataset csic 2010. URL http://www.isi.csic.es/dataset/. Accessed: 2017-12-12.

Luca Invernizzi. Support for http in scapy, 2014. URL https://github.com/invernizzi/scapy-http. Accessed: 2017-12-12.

Nayeem Khan, Johari Abdullah, and Adnan Shahid Khan. Defending malicious script attacks using machine learning classifiers. *Hindawi Wireless Communications and Mobile Computing*, 2017.

A. E. Nunan, E. Souto, E. M. dos Santos, and E. Feitosa. Automatic classification of cross-site scripting in web pages using document-based and url-based features. In *Proceedings of the 17th IEEE Symposium on Computers and Communication (ISCC '12)*, pages 702–707, July 2012.

OWASP.org. Owasp top ten project, 2017a. URL https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Accessed: 2017-12-12.

OWASP.org. Owasp top 10 – 2017, the ten most critical web application security risks, 2017b. URL https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf. Accessed: 2017-12-12.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

psy. Cross site "scripter", 2010. URL https://xsser.03c8.net/. Accessed: 2017-12-12.

K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*, pages 31–39, December 2010.

K. Schütt, M. Kloft, A. Bikadorov, and K. Rieck. Early detection of malicious behavior in javascript code. In *Proceedings of the 5th ACM Workshop on Artificial Intelligence and Security (AISec '12)*, pages 15–24, October 2012.

Symantec. Internet security threat report. Technical report, April 2016.

Various. Can be accessed at: https://pastebin.com/u6FY1xDA, https://pastebin.com/48WdZR6L, https://pastebin.com/sxxU6npD, https://pastebin.com/aiV1aP3R, https://pastebin.com/rY7mi5dT. Accessed: 2017-12-12.

Appendix A
Deployment instructions

As mentioned in the Introduction, Section 1, we get the deployment files from the `deployment` subdirectory of the GitHub repository at https://github.com/sigmundv/xss-prevent-py.

In there we find a file `docker-compose.yml`, which contains the necessary instructions to spin up PHP, nginx and MariaDB, along with CouchDB, on your server. You start this stack using the command `docker-compose up -d`, so make sure that `docker` and `docker-compose` are installed on your server.

Inside the `deployment` directory you will find a subdirectory called `public`. Download and extract the DVWA and Mutillidae applications into this directory, which will make them available. In order for these applications to work you have to change the configuration to point to database host `mariadb`. For DVWA the database user should be set to `dvwa`, and the database password ditto, in the file `config/config.inc.php`. For Mutillidae the corresponding settings are found in the file `classes/MySQLHandler.php`. Here you set the `root` password to `admin`, but if you want a better password you can change it in `docker-compose.yml` and reinitialise the mariadb container.

In order to deploy our system on your server first make sure that you have Python 3 installed (along with Python header files). After that you can install the necessary libraries by running `pip3 install -r requirements.txt` as the root user. Now you are ready to run our Intrusion Prevention System. We haven't daemonised it, so you can run it in a `screen` session, which you start with `screen -S <name of session>`. Including the name makes it easier to resume the section after you have detached from it, which would be `screen -r <name of session>`.

In the newly opened `screen` session you change to the root user and run our system with the command `python3 sniffer.py --destination <IP of the server> --port <port the web server is listening on>`.

Now you are done, and your web applications are protected to a large degree from XSS attacks.