DeeperXSS: An Exploration of the DeepXSS Approach

Matthew Bush, Dawson Brown
Ryerson University, Toronto, Canada
matthew.bush@ryerson.ca,dawson.brown@ryerson.ca

ABSTRACT

DeeperXSS is a...

KEYWORDS

Report writing, Latex tips

1 INTRODUCTION

Cross-site scripting (XSS) attacks persist as a major issue for web applications despite their root causes being well understood. The Open Web Application Security Project consistently ranks XSS in the top ten vulnerabilities on the web. For this reason, there exists a large body of work aimed at automating the detection and prevention of XSS attacks; this includes approaches using Machine Learning techniques to detect both attacks and vulnerabilities. DeepXSS is an LSTM classifier that is meant to detect XSS payloads; it was designed by Fang *et. al* and is purported to have very high precision and recall rates. For our project, we intend to recreate and extend the work of DeepXSS by Fang *et. al* [1]. We want to more clearly outline and comment on the strengths and weaknesses of their architecture as well as replicate their purported results.[1].

The primary motivation for this work is to verify Fang *et. al*'s DeepXSS method of detecting cross-site scripting (XSS) attack payloads [1]. Given that XSS is a significant problem for many web applications, the need for further research into the detection of XSS attack payloads is apparent. Unfortunately, we found that this paper lacked detail and failed to address key questions related to the work. Given DeepXSS' promising results, it would be very useful to address the lack of detail, and attempt to replicate the methods used by the authors. In verifying these results we could further our understanding of why DeepXSS was so effective (or why it was not as effective as it seemed) and apply lessons to future research.

2 BACKGROUND: DEEPXSS

This section breifly outlines the DeepXSS approach and high-lights some its shortcomings as they relate to reproducibility.

3 OUR APPROACH: DEEPERXSS

In this section we outline our approach for recreating the DeepXSS architecture, including difficulties we had in reproducibility, limitations with DeepXSS that needed to be addressed, some creative liberties we took, and a few alternative machine learning architectures that proved interesting.

Supervised by Prof. Manar Alalfi.

Program Analysis for cybersecurity(CP8320),
2021.

3.1 Data Preprocessing

3.1.1 Decoding. We built a custom recursive URL decoder. This decoder performs a depth first search of 5 different decodings. At each level, the decoder tries to decode the URL with all decodings, for each decoding that is successfull, the decoder will recursively try to further decode the string that resulted from the decoding; see algorithm 1 for a pseudocode implementation. The first string enoucountered that none of the decoders can decode is returned as the decoded string. The supported encodings are: URL unicode encoding (this includes characters of the form %uxxxx and \uxxxx),URL encoding (that is characters of the form %xx), HTML character references (that is characters of the form &xx;), hex encoding, and base64 encoding.

The major steps of the algorithm are as follows: on line 4 all the decoder functions are called and passed the URL to be decoded. These functions all return a Tuple of the form (Boolean, String) where the Boolean value indicates if the decoding was successfull while the String is the decoded string (the String remains unchanged if the decoding fails). Next, on line 8, the result tuples of all the attempted decoders are looped over. If it's found that a decoder was successful (line 10) then recurisvely call the decoder on the resulting string (line 11). If the recursive decoding succeeds than return the result of the recursive call (13). If the decoder gets to line 17, than that means one of two things happened: either none of the decoders succeeded, in which case the string must be fully decoded and so we go to line 20; if on the other hand some of the decoders succeeded, then if the algorithm gets to line 17 it must have been the case that none of the recursive calls succeeded (meaning line 13 was never reached) which means the decoder was unable to decode the string, and line 18 is run.

To clarify now what the decoder will actually do with a URL, consider the URL:

```
http://example.com/706174682F746F2F66696C653F783D26616D703B6C743B73637269707426616D703B67743B253230616C6572742825323031253230292532302616D703B6C743B2F73637269707426616D703B67743B.
```

Passing this through the decoder, the Hex decoder will succeed and produce the string:

```
http://example.com/path/to/file?x=&amp;lt;script&amp;gt;%20alert(%201%20)%20&amp;lt;/script&amp;gt;
```

the the URL decoder will succeed giving the string:

```
http://example.com/path/to/file?x=&amp;lt;script&amp;gt;alert(1)&amp;lt;/script&amp;gt;
```

then the HTML decoder will succeed and give:

```
http://example.com/path/to/file?x=&lt;script&gt;alert(1)&lt;/script&gt;
```

and finally the HTML decoder will succeed again and give:

```
http://example.com/path/to/file?x=<script>alert(1)</script>.
```

Algorithm 1 Recursive Decoder

```
1:  function DECODE(url)
2:      decoders ← [url(), unicode(), html(), hex(), base64()]
3:      dec_results ← []
4:      for decoder in decoders do
5:          dec_results.append(decoder(url))
6:      end for
7:      some_decoded ← False
8:      for decode_success, decode_str in dec_results do
9:          some_decoded ← decode_success or some_decoded
10:         if decode_success then
11:             (next_decode_success, next_decode_str) = DECODE(decode_str)
12:             if next_decode_success then
13:                 return (True, next_decode_str)
14:             end if
15:         end if
16:     end for
17:     if some_decoded then
18:         return (False, url)
19:     else
20:         return (True, url)
21:     end if
22: end function
```

Classification	Example
Start Label	<script>, <body>, <img
End Label	< /script>, < /body>
Windows Event	onerror=, onload=, onblur=, oncut=
Function Name	alert(, String.fromCharCode(
Script URL	javascript:, vbscript:
Other	>,), #

Table 1: DeepXSS Tokens.

At this point, no decoders will succeed and so the decoded string will be returned on line 20.

3.1.2 Tokenization. In DeepXSS they defined and looked for six different kinds of token summarized in table 1.

We expanded on this set of tokens and ended up with a total of 14 token types. The reason we expanded on this token set is because many many URLs especially benign URLs) contained zero tokens. For example http://www.wittebeer.be/?oid=911&pid=8056 or http://www.facebook.com/Euphnet?sk=wall, both of which are in the DMOZ directory, do not contain any of their token types. As such, we expanded their table to include 8 more as shown in table 2.

An integer argument token is any integer value that follows a function name token and is enclosed in brackets. Similarly, a string argument token is taken to be any argument that isn't an integer;

Classification	Example
Integer Argument	(543), (1), (2004)
Integer Constant	1, 2, 5432, 54
String Argument	("Hello"), (String.fromCharCode(65))
Assignment LHS	x=, variable=
Assignment RHS	=x, =654, =value
Path	path/ t56543-trer-yt43/
Identifier	iden, value, hello, goodbye

Table 2: Expanded DeepXSS Tokens.

hence *String.fromCharCode(65)* being an example of a string argument. We figured that distinguishing futher between tokens would be superfluous, and by inspection these were the most common arguments and even if the argument was a function call, the return value was almost always either an integer or a string. The integer constant token was introduced because many URLs contained nondescript integers outside of function calls and URL arguments. The two assignment tokens (left-hand side and right-hand side) were introduced so that URL arguments would not be lost (notice that the two benign examples include arguments that are skipped by DeepXSS). Similarly, path tokens were intoduced so that the path wasn't entirely skipped. And finally we introduced a generic identifier token. Assignment LHS, Path, Script URL, Function Name, and Windows Events all begin with an identifier but all have additional characters to further distinguish them (for example Windows evens always start with 'on' and end with '=', paths end with '/' or '?', etc). Any identifier that is found that cannot be further categorized is included as a generic Identifier token.

3.1.3 Generalization. In keeping with DeepXSS [1], we generalized many parts of the URL. In the original paper all the domains became simply *website*. In our case, we simply ommitted the domain since including *website* for every domain communicates no information. We mapped all integer arguments to '(1)' and all string arguments to '("str_arg")'. All integer constants were mapped to 1. The left-hand side of assignments were all generalized to 'assign<num>=' where '<num>' starts from 0 and increments for each encountered left-hand side. This was done because most identifiers in URLs proved to be unique, and so not useful in classification (especially when using CBOW) with the exception of Function Names, script URLs, and Windows Events which are reserved words and so can't be unique across many URLs. The right-hand sides also took the form 'val<num>' where '<num>' again increments with each right-hand side encountered. Similarly, all path tokens were mapped to 'path<num>/' where '<num>' again starts at 0 and increments with each path token that's found. Finally all identifier tokens were mapped to 'ident<num>'.

3.1.4 An Example. To take for example `http://website/search.php?uid=ws848d8024088c327.36898031&src=&term=<script>alert(123)</script>&args=qs=06o` the following sequence

Model	Output	Embedding	Input
Value-Softmax	softmax	CBOW	Token values
Value-Sigmoid	sigmoid	CBOW	Token values
Value-Sequential	softmax	None	Token values
Value-Random	softmax	Random	Token values
Type-Softmax	softmax	CBOW	Token types
Type-Sigmoid	sigmoid	CBOW	Token types
Type-Sequential	softmax	None	Token types
Type-Random	softmax	Random	Token types

Table 3: Summary of Models

Prediction	XSS	Not XSS
Predicted XSS	t_p	f_p
Predicted Not XSS	f_n	t_n

Table 4: Confusion Matrix

of (token type, token value) pairs would be extracted going left to right:

(1) `(Path, 'path0/')` for search.php?
(2) `(Assignment LHS, 'assign0=')` for uid=
(3) `(Assignment RHS, 'val0')` for =ws848d8024088c327
(4) `(Integer Constant, 1)` for 36898031
(5) `(Assignment LHS, 'assign1=')` for src=
(6) `(Assignment LHS, 'assign2=')` for term=
(7) `(Start Label, '<script>')` for `<script>`
(8) `(Function Name, 'alert')` for alert(
(9) `(Integer Argument, '(1)')` for (123)
(10) `(End Label, '</script>')` for `<script>`
(11) `(Assignment LHS, 'assign3=')` for args=
(12) `(Assignment LHS, 'assign4=')` for qs=
(13) `(Integer Constant, '06')` for 06
(14) `(Identifier, 'o')` for o

Note that if two different token types overlap, the one that starts earlier is preferred; if they start in the same place then the one that extends further is preferred. This is why 'args=qs=' is taken to be two left-hand sides–the 'qs' could be a right-hand side, but with the equals on the right its taken to be a left-hand side since that is longer and both start at the 'q'. As per the last two tokens, identifiers cannot start with numbers (in Javascript) and so the 06o is taken to be an integer constant and an identifier.

Model	Precision	Recall	F1	Accuracy
DeepXSS	0.995	0.979	0.987	n/a
DeeperXSS:softmax	0.989	0.973	0.981	0.981
DeeperXSS:sigmoid	0.988	0.976	0.982	0.983
DeeperXSS:sequence	0.991	0.956	0.973	0.975
DeeperXSS:random	0.099	0.097	0.098	0.56

Table 5: Model Comparison.

3.2 Word2Vec

3.3 LSTM Classifier

4 EVALUATION

$$\text{Precision} = \frac{t_p}{t_p + f_p}$$

$$\text{Recall} = \frac{t_p}{t_p + f_n}$$

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Accuracy} = \frac{t_p + t_n}{t_p + t_n + f_p + f_n}$$

5 RELATED WORK

Mokbal *et al.* created a multilayer preceptron (MLP) model for detecting XSS both in dynamic webpages and URLs [3]. Their approach, called MLPXSS, has three main pillars: data scraping, feature extraction, and an artificial neural network ANN. Their model is meant to deal with both dynamic webpages and malicious URLs. Their feature extraction level has 3 modules to extract HTML-based features, Javasript-based features, and URL-based features. The HTML module tokenizes tags, attributes and events–focusing on things that trigger Javascript execution (like *href* or *onclick*). The Javascript module parses and tokenizes Javascript code that is pulled from a webpage. There are various ways to include Javascript in a page like script tags, *onclick* and *onsubmit* calls, *href*, etc... Lastly they tokenize potentially malicious parts of URLs, like HTML properties, tags, some keywords (*login*, *signup*), *document* references, and various special characters like '<', '>' and '/'. The MLP is trained on token streams with a sigmoid output layer. Their perceptron had precision, f-measure, and accuracy all in excess of 99% [3].

Zhang *et al.* propose a dual Gaussian mixture model (GMM) approach that trains two seperate GMMs models (one for benign and one for malicious) and then combines their outputs to make a prediction. Additionaly, they train the models on both the URL and the server response to the URL in an attempt to get richer features [5]. To preprocess the URLs, the decode, tokenize, and train a word2vec model to retrieve a vector representation of each token.

Their tokenization approach is very similar to that of DeepXSS and MPLXSS, they however inclue the domain and path for the benign GMM. They are however generalized to simply ‘domain’ and ‘path’ [5]. Their reasoning for this is that containing just a domain and path is characteristic of a benign URL, whereas an XSS URL is characterized by its maliciouly constructed parameters and not the presence of a domain and path [5]. Their models can be trained on requests, responses, or both. They reason that in many cases, benign requests contain no XSS tokens, which isn’t very interesting, however responses contain useful features for both XSS and none-XSS tokenization. Their multi-stage dual GMM using both responses and requests greatly improved classification [5].

Goswami *et al.* propose a attribute clustering technique to perform unsupervised grouping of malicious and benign scripts. They’re feature extraction is wholly different from DeepXSS and other deep learning classifiers. They propose 15 features that characterize malicious and benign scripts creating a 16-dimensional vector for each script (including class) [2]. These features are meta-features like length of the script, number of strings and the average string length, number of methods, number of unicode and hex characters, among others. These features are then min-max normalized before clustering [2]. Their algorithm was able to achieve an accuracy in excess of 98% [2].

6 DISCUSSION

This approach is entirely dependent on the scope and correctness of the decoder. The filter evasion techniques used for XSS can be extremely complicated and can include multiple and mixed encodings with near arbitrary white space [4]. New evasion techniques are constantly being discovered as well. The decoder has to be sophisticated enough to handle the evasion techniques of novel XSS payloads to be useful in practice. That said, a good decoder will be hard to evade and probably only a negligible number of XSS payloads will get past it, on the other hand, given the severity of the consequences of some XSS attacks, no number is negligible.

As with all machine learning, this approach is very sensitive to the preprocessing of the data and the feature extraction. In our case, this is the tokenization step. There have been many different tokenization procedures proposed; often they exclusively looked for tokens that would indicate a malicious URL and largely ignored the benign segments of a URL [1][3]. This approach means that many benign URLs contain no tokens [5]. This sort of approach requires the designer of the tokens to judge what kinds of strings characterize malicious URLs and only tokenize those which can be hard to do well. To address this, perhaps both the URL and the servers response to it can be tokenized which gives a much richer set of features and would help fill out the empty URLs [5]. This has the draw back of having interact with the server for each URL which adds overhead–ideally the URL provides sufficient information. In our approach, we had a more general tokenization approach that aimed at tokenizing URLs and not just malicious URLs. The idea being that the machine learning algorithm can be left to figure what sort of strings and patterns characterize malicious URLs without much input from us. This means benign URLs aren’t empty and the server response is not required. That said, there are likely benefits to including special strings that give a strong indication of an XSS payload.

7 CONCLUSION

REFERENCES

[1] Yong Fang, Yang Li, Liang Liu, and Cheng Huang. 2018. DeepXSS: Cross site scripting detection based on deep learning. In *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*. 47–51.
[2] Swaswati Goswami, Nazrul Hoque, Dhruba K Bhattacharyya, and Jugal Kalita. 2017. An Unsupervised Method for Detection of XSS Attack. *Int. J. Netw. Secur.* 19, 5 (2017), 761–775.
[3] Fawaz Mahiuob Mohammed Mokbal, Wang Dan, Azhar Imran, Lin Jiuchuan, Faheem Akhtar, and Wang Xiaoxi. 2019. MLPXSS: an integrated XSS-based attack detection scheme in web applications using multilayer perceptron technique. *IEEE Access* 7 (2019), 100567–100580.
[4] OWASP. 2021. XSS Filter Evasion Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html
[5] Jingchi Zhang, Yu-Tsern Jou, and Xiangyang Li. 2019. Cross-site scripting (XSS) detection integrating evidences in multiple stages. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*.