DeeperXSS: An Exploration of the DeepXSS Approach

Matthew Bush, matthew.bush@ryerson.ca
Dawson Brown, dawson.brown@ryerson.ca

Ryerson University

Abstract. DeeperXSS is a...

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

[1]

2 DeepXSS

3 DeeperXSS: exploring DeepXSS

3.1 Data Preprocessing

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.

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
```

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

http://example.com/706174682F746F2F66696C653F783D26616D703B6C74
3B7363726970742661 6D703B67743B253230616C65727428253230312532302925
323026616D703B6C743B2F7363726970742661 6D703B67743B.

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;%20aler
t(%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;/sc
ript&gt;

and finally the HTML decoder will succeed again and give:

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

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

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

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

Table 1. DeepXSS Tokens.

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.

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

Table 2. Expanded DeepXSS Tokens.

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; 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 non-descript 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.

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¿'.

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 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.

3.2 Word2Vec

3.3 LSTM Classifier

3.4 Evaluation

4 Comparison

5 Related Work

6 Discussion and Conclusion

References

1. Fang, Y., Li, Y., Liu, L., Huang, C.: Deepxss: Cross site scripting detection based on deep learning. In: Proceedings of the 2018 International Conference on Computing and Artificial Intelligence. pp. 47–51 (2018)