VDPython

A VulDeePecker algorithm implementation in Python

Introduction

Security vulnerabilities are widespread problem that can have major consequences if exploited. Experts in information security can usually identify potentially exploitable pieces of code, but having experts available is not always feasible nor necessarily sufficient to make a program secure. Alternatively, automated vulnerability detection would alleviate the need for expertise. Existing solutions approach the problem through static code analysis or machine learning on manually defined features of the code. These methods can help, but usually result in either too high false positive rate (code is flagged when it is safe), or too high false negative rate (exploitable code is deemed safe). Both approaches also require a security expert to produce static rules or meaningful features, a process that can be time consuming. An alternative approach would be to apply deep learning to the problem.

Deep learning has a major advantage due to the fact that features do not need to be defined by humans. This is useful for automation purposes without the need of an expert, and also helps produce more effective systems. However, programs first need to be represented in a way that a neural network could work with. We also need to represent programs in meaningful chunks that are checked for vulnerabilities.

Data

For training the model and testing new code, programs are converted into code gadgets. These gadgets are semantically related chunks of code. These chunks can be vectorized for use by the neural network. The dataset being used is the same dataset provided by the original creators of the VulDeePecker system. It contains 61,638 gadgets, with 10,440 of those gadgets containing a buffer error vulnerability, and 7,285 gadgets containing a resource management error. For reporting accuracy scores of our model, we split the data 80% for training and 20% for testing.

Implementation

Modularization

A beneficial technique when developing a project as a group is modularization, where the tasks are broken up into independent modules, and each module has one or more standalone functions that aren't dependent on other modules. This both enables parallel development and prevents interference between modules. This would be optimal for a group project like this one, especially whose team members cannot regularly meet and would be working alone for the majority of the project. It also cuts down development time, as each module can be completed simultaneously with minimal modification when combining.

This project was divided into 4 coding modules: converting gadgets to symbolic representation (cleaning), "vectorizing" gadgets, constructing a BLSTM neural network for training and testing, and an interface to the project that will combine all the modules. The first module involved a step in the algorithm where the names of user-defined variables and functions are replaced with VAR# and FUN#, respectively. The "#" is a placeholder for some integer identifier of a particular variable or function within a gadget. This allows a major reduction when converting to tokens. The second step involved "tokenizing" and "vectorizing" each gadget. Tokenizing refers to breaking down a gadget into an array (or vector) of tokens, where each token is a symbol, keyword, or operator.

The technique Word2Vec is then used to create word embeddings for each token, and these embeddings are combined to create a vector for each gadget. These vectors are then used as input to the BLSTM neural network. The last module, the project interface, extracts the gadgets from a gadget file, uses the symbolic representation module, uses the output of that as the input to the vectorizing module, and finally combines the gadget vectors and their vulnerability indicators into a data structure to be passed to the neural network.

All modules and their implementations are explained in greater depth below.

Transforming Code Gadgets into Vectors

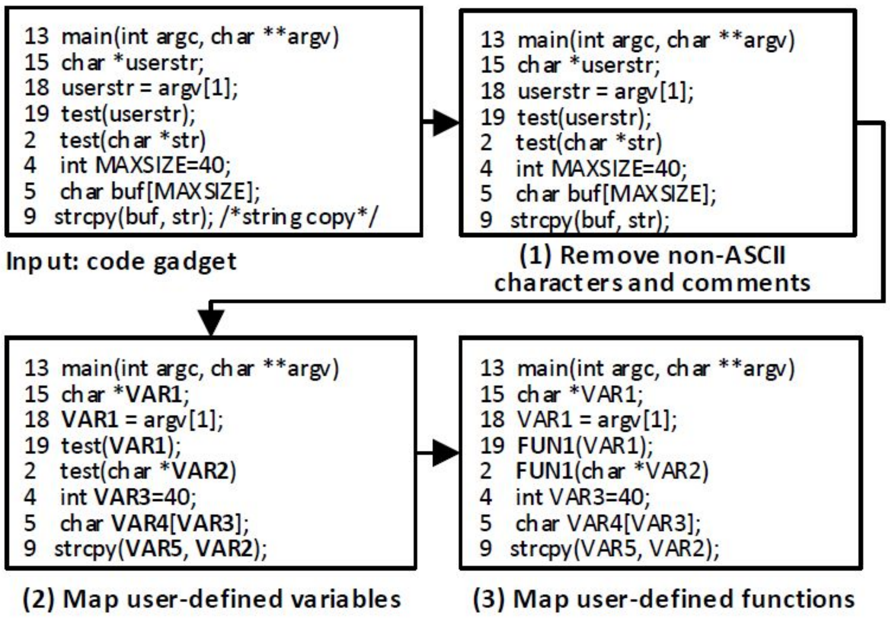
Transforming Code Gadgets into Their Symbolic Representations

This step aims to heuristically capture some semantic information in the programs, with the goal of transforming code gadgets into vector inputs for neural network training. The identifiers of user-defined variables and user-defined functions are transformed into symbolic names, while any *main* function identifier and its arguments remain untouched. C and C++ keywords also remain unchanged. Thus, our code includes immutable sets to contain the *main* identifier, it's common arguments (*argc*, *argv*), and a complete collection of keywords up to C11 and C++17.

Our input is a list of strings (lines), representing a complete code gadget. For each line, we remove string literals and character literals (as they do not figure into potential vulnerabilities), while keeping the quotes. For the same irrelevancy, we also delete any code comments and non-ASCII characters that may be present.

The user-defined variables are mapped to the symbolic names, "VAR1", "VAR2", etc. User-defined functions are mapped to "FUN1", "FUN2", etc. These mappings are performed in a one-to-one fashion. The symbolic name counts are reset per code gadget, so multiple variables and functions may be mapped to the same symbolic name when they appear in different gadgets.

We use regular expressions to capture candidate identifiers. A candidate identifier that is not included in one of the immutable sets will be replaced by a symbol. To retain semantics, order is maintained internally within the line and externally across the code gadget. The result is a "cleaned gadget," which is the output for the tokenization and vectorization processes.


Input: code gadget
(1) Remove non-ASCII characters and comments
(2) Map user-defined variables
(3) Map user-defined functions

```
13 main(int argc, char **argv)
15 char *userstr;
18 userstr = argv[1];
19 test(userstr);
2  test(char *str)
4  int MAXSIZE=40;
5  char buf[MAXSIZE];
9  strcpy(buf, str); /*string copy*/
```

```
13 main(int argc, char **argv)
15 char *userstr;
18 userstr = argv[1];
19 test(userstr);
2  test(char *str)
4  int MAXSIZE=40;
5  char buf[MAXSIZE];
9  strcpy(buf, str);
```

```
13 main(int argc, char **argv)
15 char *VAR1;
18 VAR1 = argv[1];
19 test(VAR1);
2  test(char *VAR2)
4  int VAR3=40;
5  char VAR4[VAR3];
9  strcpy(VAR5, VAR2);
```

```
13 main(int argc, char **argv)
15 char *VAR1;
18 VAR1 = argv[1];
19 FUN1(VAR1);
2  FUN1(char *VAR2)
4  int VAR3=40;
5  char VAR4[VAR3];
9  strcpy(VAR5, VAR2);
```

Decomposing the Symbolic Representations into Tokens

The cleaned gadget obtained from the previous step is processed to convert each line of the gadget into an array of tokens. The tokens consist of identifiers, keywords, operators and symbols. Words are added to the list whenever they are not a part of the set of operators or symbols. For example, a line provided as a string,
“unsigned char *ret = p;”
Gets converted into,
[“unsigned”, “char”, “*”, “ret”, “ =”, “p”, “;”]
The resulting list of token is used to form Vectors.

Encoding the Tokens into Vectors

The previous tokenization step converted each line in a gadget into tokens, giving a concatenated array of tokenized lines for each gadget. This data is then directly inserted into a Word2Vec model, which was implemented in the gensim Python module. The Word2Vec model accepts tokenized sentences as input, so gadgets were treated as sentences, and keywords, operators, and symbols were treated as “words” (tokens). The Word2Vec model must be trained on all gadgets in the file, so the tokenized gadgets were buffered into memory and then the model was trained on all gadgets at once.

Since Word2Vec creates a word embedding (vector) for each token, the vectors are 2D, essentially a collection of 1D vectors, where each 1D vector is a token embedding. Not specified in the paper is whether a skipgram or continuous Bag-of-Words (BOW) model for Word2Vec is used, so both were tried with similar results. The shape of the 2D vectors is pre-chosen. The size of each 1D vector is the size of each word embedding, which is fed into the Word2Vec model. This is 100 by default for Word2Vec, but this led to large vectors, so 50 was used instead. The number of embeddings chosen per gadget was 50, which was taken from the paper. For gadgets with less than 50 tokens, these arrays were padded with 0's, and for gadgets with more than 50 tokens, the arrays were truncated. Depending on whether or not a gadget was take from a forward slice or backward slice(s), the back or front of the gadget vector was padded/truncated, respectively. According to the paper, library/API calls are placed in the last line of the a backward slice gadget. As it happens, all of those calls would be transformed to "FUN#" in the "cleaning" stage. Therefore, to detect whether a gadget is a forward/backward slice, the symbol "FUN" was searched in the last line of the gadget; if found, then the gadget is determined to be a backward slice, but if not, then a forward slice.

Training the Neural Network

The preprocessed data was strongly imbalanced in favor of negative examples for both the buffer error vulnerability dataset and the resource management error vulnerability dataset, which can cause the majority class to be favored heavily by the loss function. We first attempted to simply weight the classes so that the loss function assigned proportional importance to the positive and negative examples. Finding this to be lacking, we randomly undersampled the negative class such that we had equally as many negative examples as we did positive. Because of the nature of Word2Vec embeddings and the ordered nature of code gadgets, we thought it would be inappropriate to attempt to oversample the positive examples. This resampled dataset was then split into 80% train and 20% test in a stratified manner.

The network architecture was chosen to match the architecture suggested by VulDeePecker as closely as possible. Unfortunately, the final architecture used to generate the results given in the paper was not fully described. We know from their description that the input layer was a bidirectional long short term memory layer, and that the final layer was a softmax layer. We likewise know that they ran experiments varying the number of fully connected hidden layers, with the best results at two or three hidden layers. However, the number of hidden layers and their activation functions used to generate their results was not given. For this reason, we chose to use two fully connected hidden layers with LeakyReLU as their activation function. LeakyReLU was chosen because ReLU is an appropriate and robust activation function for many domains, but can suffer from the dying ReLU problem wherein a node is overwhelmed by a sufficiently large gradient and only emits 0 from that point forward. A dropout layer follows both hidden layers.

Hyperparameters were also chosen to match those suggested by VulDeePecker. Where possible, default hyperparameter values were used. VulDeePecker explicitly notes that their dropout was 0.5, their batch size was 64, their number of epochs was 4, their number of nodes per layer was 300, and they optimized with ADAMAX. We deviated from their implementation by using a learning rate of 0.002 compared to their 1 as we found that their learning rate was excessive. While not explicitly stated, we are relatively certain that VulDeePecker also used categorical cross-entropy as their loss function.

Results

The results are split based on the performance on the two different vulnerabilities: buffer errors and resource management errors. We also report the results from the original VDP paper as a point of comparison.

For buffer errors, our model got a False Positive Rate (FPR) of 37%, False Negative Rate (FNR) of 8.3%, and an F1 score of 80.2%. By comparison, the original got 2.9%, 18%, and 86.6% in each respective metric. Our model was able to achieve a low FNR, meaning that a large number of the vulnerabilities in the test data set were caught. The original paper's model had a better false positive rate, meaning that our model declare more safe gadgets as exploitable than the original.

For resource management errors, our model got an FPR of 15%, FNR of 4.2%, and F1 score of 90.9%. The original got 2.8%, 4.7%, and 95% in each respective metric. As with buffer errors, the low FNR means that most of the vulnerabilities were caught. Also, the FPR is much lower for this kind of error, making the F1 score higher.

Overall, our model's ability to detect errors is very good. However, the model is the paper is better at correctly declaring gadgets as safe. depending on the desired FPR, either model could be used to detect a large amount of vulnerabilities in software.