12-2018

Learning-based Analysis on the Exploitability of Security Vulnerabilities

Adam Bliss
University of Arkansas, Fayetteville

University of Arkansas

Learning-based Analysis on the Exploitability of Security Vulnerabilities

by

Adam Bliss

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Bachelor of Science in Computer Science with Honors

APPROVED, THESIS COMMITTEE

Qinghua Li, Ph.D.
Advisor and Committee Chair

Matthew Patitz, Ph.D.
Committee Member

Dale Thompson, Ph.D.
Committee Member

FAYYETEVILLE, ARKANSAS
November 2018

Abstract

The purpose of this thesis is to develop a tool that uses machine learning techniques to make predictions about whether or not a given vulnerability will be exploited. Such a tool could help organizations such as electric utilities to prioritize their security patching operations. Three different models, based on a deep neural network, a random forest, and a support vector machine respectively, are designed and implemented. Training data for these models is compiled from a variety of sources, including the National Vulnerability Database published by NIST and the Exploit Database published by Offensive Security. Extensive experiments are conducted, including testing the accuracy of each model, dynamically training the models on a rolling window of training data, and filtering the training data by various features. Of the chosen models, the deep neural network and the support vector machine show the highest accuracy (approximately 94% and 93%, respectively), and could be developed by future researchers into an effective tool for vulnerability analysis.

1 Introduction

As computer systems grow increasingly complex, significantly more advanced techniques are needed to keep them secure. This becomes especially apparent at an enterprise level, where system administrators face the daunting challenge of securing a network of hundreds or thousands of connected devices. If a single one of these systems were to be compromised, it could spell disaster for the entire enterprise.

One of the most obvious steps an administrator can take to secure their network is keeping their systems up-to-date with the most current patches from their software vendors. However, this is a much more complex task than it immediately appears. A large-scale enterprise can consist of hundreds or thousands of work stations, mobile devices, servers, storage devices, high-performance computers, and other infrastructure. Across these various types of devices, dozens or hundreds of distinct services are running: file transfer services, mail servers, web servers, databases, remote access tools, field devices, and more. Each of these services has the potential to contain one or more vulnerabilities that could be used to cause devastating harm to a system and potentially the entire organization.

Although keeping these sprawling systems patched is a formidable challenge, it is one that cannot be neglected. A painfully obvious example of the consequences of such neglect is the 2017 WannaCry ransomware attack. Over the course of only a few days, this malware spread to over 230,000 computers across 150 countries [3]. This attack caused incalculable damage, affecting multiple sectors: energy, transportation and logistics, communications, shipping, and most prominently, health care. Britain’s National Health Service reported outages in hospital computers, MRI equipment, cold-storage facilities, and more. Thousands of medical appointments, some urgent, had to be canceled or rescheduled; operations had to be postponed or relocated; patients were diverted away from affected facilities until order could be restored.

This single vulnerability laid bare some of the nation’s most sensitive equipment for a devastating attack – but it was preventable. Only a few short months before, Microsoft had released a patch for the affected vulnerability, describing it as “critical” and recommending the affected parties update their software immediately [4]. Those who heeded the warning from Microsoft escaped the WannaCry attack unscathed; those who failed to listen suffered. The message is sobering, but simple: security patches cannot be ignored.

Most patches are aimed at fixing one or more known vulnerabilities, flaws in software that cause a security risk. With almost 9,000 new and unique vulnerabilities reported in 2017 alone, keeping these vulnerabilities patched becomes a daunting challenge [1]. However, not all vulnerabilities are created equal: some are much more likely to be used in an exploit, and these are the vulnerabilities that should be patched first. This is particularly important for organizations with limited resources that might struggle to handle vulnerabilities and patches quickly. For example, many mid-size and small electric utilities do not have sufficient security operators to address vulnerabilities and patches in a timely manner. System administrators need a tool that can accurately classify known vulnerabilities by their likelihood of being used in an exploit.

But how should such a tool be created? Several requirements must be considered when designing this tool. It must be accurate: if the tool fails to provide reliable recommendations, it will not be trusted and will not help prevent future attacks. It must be adaptable: the landscape of threats facing large enterprises is constantly shifting, and any tool attempting to mitigate these threats needs to be able to adapt with them. With these considerations in mind, a learning-based approach is chosen for development. Using machine learning techniques allows the development of a tool that draws from the vast amount of already-available data in order to perform the most accurate analysis possible of new vulnerabilities. Furthermore, such a tool can be periodically retrained on new vulnerability data, ensuring it stays reliable as the types of threats evolve.

The objective of this thesis is to use machine learning techniques to create a tool that analyzes known vulnerabilities and classifies them according to their exploitability. In order to ensure the greatest possible accuracy, several different machine learning models (specifically, a deep neural network, a random forest classifier, and a support vector machine) will be implemented and compared. The most accurate model will be built into a tool which can later be integrated into an enterprise-wide patch management system that keeps track of all current vulnerabilities present in an enterprise and ranks them according to their exploitability. This will allow system administrators and security specialists to determine which vulnerabilities are most likely to be used in an exploit and consequently, which ones to patch first. By allowing administrators to intelligently patch their systems, successful exploits can be kept to a minimum and attacks such as the WannaCry ransomware attack can be prevented.

In addition to describing the design and implementation of a learning-based tool for vulnerability analysis, this report will cover some important background on the topic, including the core concepts behind security patch management and the underlying structure of several learning-based models. Furthermore, this report contains a discussion on related work, including conventional vulnerability analysis techniques and other implementations of various learning-based vulnerability analysis tools.

2 Background

2.1 Related Work

Using machine learning techniques to create advanced security tools is not a new concept. Learning-based systems have been proposed for everything from spam detection to network intrusion detection [2]. If sufficient training data exists, these algorithms can spot patterns and trends too subtle for human analysts, potentially giving security technicians a much-needed edge against sophisticated attackers. But before these learning-based approaches are discussed, this report will review conventional techniques for vulnerability analysis.

2.1.1 Conventional Vulnerability Analysis

Conventional vulnerability analysis can be broadly broken down into three main methods: (1) static analysis, (2) dynamic analysis, and (3) hybrid analysis [6]. A discussion of each of these methods follows:

(1) **Static analysis:** a program's source code is directly analyzed, without executing any part of the program.

(2) **Dynamic analysis:** a program is executed and the runtime behavior is monitored and analyzed.

(3) **Hybrid analysis:** a program is analyzed with a mixture of static and dynamic analysis techniques.

Each of these methods have their respective strengths and weaknesses. Static analysis uses a generalized abstraction to analyze the properties of the program, which has the potential to be sound (i.e., potentially no missed vulnerabilities), but also a possibility of generating false positives. In contrast, dynamic analysis is at best complete (i.e., no false vulnerabilities), but cannot be sound, because the potentially infinite amount of possible inputs to a program mean that there is always a possibility for vulnerabilities to exist in an unseen program state. Hybrid analysis combines the strength of both these approaches, but combines their weaknesses as well. Typically, a hybrid analysis is either a static analysis system that utilizes dynamic analysis to help identify and reduce false positives, or a dynamic analysis system that uses static analysis techniques to help reach hidden program states and potentially uncover missed vulnerabilities.

In addition to these conventional program analysis techniques, several so-called discovery approaches exist: (1) software penetration testing, (2) fuzz testing, and (3) static data-flow analysis.

(1) **Software penetration testing:** a team of software security experts manually test and probe a program for exploitable vulnerabilities [7].

(2) **Fuzz testing:** similar to dynamic analysis, manual program inputs are created. These are then randomly mutated and fed into the program on a large scale in an attempt to reach hidden program states [8].

(3) **Static data-flow analysis:** untrusted data from input sources is marked as tainted and its flow to program states known as sinks is monitored as a potential indicator of vulnerabilities [9, 10].

Together, these program analysis and discovery methods make up the bulk of the methodology for conventional vulnerability analysis. In the next section, the currently accepted techniques for machine learning-based analysis are discussed.

2.1.2 Learning-based Analysis

In recent years, researchers have been exploring a new approach to vulnerability analysis: machine learning and data mining. Arthur Samuel, the scientist who coined the term “machine learning” in 1959, defined it as the development of computational techniques and algorithms that enable computer systems to gain new abilities without being explicitly programmed [11]. This ability to develop behavior without being explicitly programmed ahead of time is of obvious value in the field of data mining, defined as the computational process of extracting actionable knowledge from large amounts of data, including data extraction, data cleansing, data selection, knowledge extraction, and visualization [12]. Machine learning can be extremely useful in the data extraction, cleansing, and selection stages of the data mining process, as the ability to "train" a model on a vast data set allows the machine to detect patterns and trends not visible to human analysts.

Machine learning techniques can be grouped into three main methods: (1) supervised learning, (2) unsupervised learning, and (3) reinforcement learning [5].

(1) **Supervised learning:** requires a set of labeled training data, which consists of a set of records, where each record contain some input data (usually in a vector format) and a desired corresponding output label. The model is trained on this data, and uses the provided labels for validation and testing.

(2) **Unsupervised learning:** primarily used in situations where labeled training data is not available. Rather than iteratively train and refine itself, the learning system attempts to identify patterns and trends in the provided dataset.

(3) **Reinforcement learning:** the learning system is trained in the optimal method of achieving a certain goal by a system of receiving rewards and penalties for its actions in a dynamic environment.

These are all approaches for machine learning in general, not specific to vulnerability analysis. Several specialized techniques have been pioneered to apply these machine learning techniques to more intelligently analyze vulnerabilities. These techniques include: (1) vulnerability prediction models based on software metrics, (2) anomaly detection approaches, and (3) vulnerable code pattern recognition. The first method is most closely related to the work described in this report, and will be the focus of the remainder of this section.

This method involves utilizing a supervised machine learning approach to build a prediction model based on well-known software metrics as the feature set [5]. Essentially, this type of analysis attempts to predict vulnerable software components (such as binary components, object-oriented classes, source code files, etc.) based on common software engineering metrics. By compiling a set of training data from various programs, a model can be trained to identify which software components are most likely to contain a vulnerability. This is similar to the work described in this thesis, which focuses on an entire set of vulnerabilities rather than a specific software suite. By compiling a set of training data from open-source repositories, a model can be trained that recognizes which vulnerabilities are more open to exploitation so that system administrators can patch the affected software systems and prevent a damaging breach of the system.

3 Design

The goal for this thesis is to predict the exploitability of vulnerabilities. In this case, a prediction is defined as a binary decision: will a given vulnerability ever be used in an exploit or not? Although this is a rather simplified answer to a very nuanced question, the models designed and implemented in this thesis provides a foundation for future researchers to develop a tool that provides a more sophisticated answer to this question.

The basic intuition behind the learning-based approach to this thesis is that similar vulnerabilities would have similar exploitability. Therefore, if a model is trained on past vulnerabilities, it might be able to recognize similar vulnerabilities based on the selected features and use this similarity to make a prediction on the exploitability. The process of selecting and curating these features is discussed in section 3.1.

As previously mentioned, several different models are implemented as part of this thesis. These models (specifically, a deep neural network, a random forest classifier, and a support vector machine) will be discussed in greater detail in section 3.2. The architecture of the deep neural network requires several additional design decisions, and these will be discussed in section 3.3.

3.1 Data Sources

One of the most important elements of this thesis is the process of selecting and curating a comprehensive set of training data from multiple open-access sources. A machine learning model is only as good as the data used to train it; therefore, compiling a useful set of data is critical. Several online databases are used for the creation of the training data, and are chosen for both the relevance and accessibility of their data. Both databases make their data available in an easily-accessible XML data feed, making them particularly easy to parse into a single set of training data. The two sources chosen are the National Vulnerability Database and the Public Exploit Database, and the structure and content of the data from these sources is explained in the following subsections.

3.1.1 National Vulnerability Database and Feature Selection

The National Vulnerability Database (NVD), published by the Information Technology Laboratory of the United States National Institute of Standards and Technology (NIST), is the official U. S. government repository of standards-based vulnerability management data [13]. The data is represented using the Security Content Automation Protocol, which allows the data to be used in automation of vulnerability management, security measurement, and compliance. It contains a wealth of data on tens of thousands of discrete vulnerabilities, including impact assessments, attack vectors, classifications, affected software, and more. The fields chosen as features for the dataset are listed below:

- **Access Vector:** the vector through which the vulnerability allows access to the system. Possible values: Network, Adjacent, Local, Physical.
- **Access Complexity:** the complexity of the access allowed by the vulnerability. Possible values: Low, High.
- **Confidentiality Impact:** potential impact of the vulnerability on the confidentiality of the system. Possible values: None, Low, High.
- **Integrity Impact:** potential impact of the vulnerability on the integrity of the system. Possible values: None, Low, High.
- **Availability Impact:** potential impact of the vulnerability on the availability of the system. Possible values: None, Low, High.
- **CVSS Score:** Common Vulnerability Scoring System score, a common framework for ranking potential impact of vulnerabilities. Possible values: 0.0-10.0.

These fields were primarily chosen for the ease of encoding them as numerical values that could be used as features for the model (this encoding is described in section 4.1.1). The possibility of adding more features from the database is discussed in the ‘Future Work’ section of the conclusion. As previously mentioned, all of the records in the database are available for downloading as an XML data feed, which makes it ideal for locally mirroring and processing into a format suitable for training a machine learning model.

3.1.2 Offensive Security Exploit Database

The second database used in the compilation of the training data is the Exploit Database (Exploit-DB) published by Offensive Security, an information security training company that provides information security certifications as well as penetration testing services [14]. Unlike the NVD, the Exploit-DB does not provide features for use of training the model – it supplies the labels used to curate a set of training data suitable for supervised learning. Without the labels provided by the Exploit-DB, a supervised learning approach (such as a DNN) would not be possible. The NVD provides the information on the vulnerability, but that information is not useful unless it can be correlated with the existence (or lack of existence) of an exploit using that vulnerability. By providing a method to cross-reference the vulnerability from the NVD with a known exploit from the Exploit-DB, a training set can be created that contains, for each vulnerability, the information on that vulnerability and whether or not that vulnerability has appeared in a known exploit. This provides a complete set of labeled training data that can then be used in a supervised learning approach.

3.1.3 CVE Reference Map for Exploit-DB

The method for correlating the two databases is actually provided by the same organization that provides the NVD. NIST maintains a framework for identifying vulnerabilities known as the Common Vulnerabilities and Exposures (CVE) list. This framework is used to provide each discrete vulnerability with an ID number, beginning with “CVE-XXXX-YYYY”, where XXXX is the year the vulnerability was recorded and YYYY is the number of the specific vulnerability [15]. CVE also provides a set of reference maps, which correlates CVE ID numbers to ID numbers from other well-known services. One of the reference maps is for the Exploit-DB, which provides a framework for matching known vulnerabilities with known exploits to create the training data. Unfortunately, the reference map is not available in XML format, so a Python script was written (using the libraries Requests and BeautifulSoup) to scrape and parse the reference map into a usable format. Once this was done, a second script was used to iterate the contents of the NVD data feed, parse the relevant features (discussed in detail in section 4.1.1), and check the saved reference map for the existence of a correlated exploit. Although the reference map was sorted prior to being mirrored locally, this is still a computation-expensive process. However, runtime is not a pressing concern, as this process only has to be done when additional training data is required.

3.2 Machine Learning Models

As previously mentioned, several different types of machine learning models will be implemented as part of this thesis. The three models selected for experimentation are: deep neural networks (DNN), random forest classifiers, and support vector machines (SVM). These models are selected because of their applicability to this type of problem; specifically, all three of these models are considered suitable for a classification problem (grouping observations into sub-groups) [24]. The following subsections contain a discussion of each of the three models selected, including their structure, function, and weaknesses.

3.2.1 Deep Neural Networks

A deep neural network is a specialized form of an artificial neural network (ANN), a type of computing system designed to roughly emulate the biological neural networks present in animal brains [21]. An ANN is composed of connected nodes (neurons), where each connection can transmit a signal from one node to another. The receiving neuron can then process the signal and transmit the revised signal to another neuron. Each neuron and connection (edge) has a weight, a value which is adjusted as the model is trained. The neurons are typically arrayed in a series of layers, with the signal traveling from the first layer (input layer) to the last layer (output layer). A DNN is a specific form of an ANN which contains multiple hidden layers, layers between the input and output layers.

DNNs are well-suited for classification problems, due to their ability to classify nonlinear input [25]. However, DNNs are not perfect: potential drawbacks include overfitting on the training data and computation time. Overfitting is an issue where the model fits too closely to the provided training data, and fails to model a real-world scenario (performance drops significantly when introduced to new data) [19]. Measures can be taken to prevent overfitting, including limiting the number of hidden layers and manipulating the training data, but it remains a present concern for DNN usage. Second, training a DNN is a time- and computation-heavy process, especially when using hundreds of thousands of data points (as recommended for higher accuracy) [26]. Several techniques can be used to reduce this time, such as batching (computing the gradient on several records at once rather than individually), but it is still a concern, especially in scenarios where limited computation resources are available.

3.2.2 Random Forest Classifiers

A random forest (or random decision forest) is a method of ensemble learning (a type of learning that uses multiple algorithms to obtain more accurate predictions) that utilizes multiple decision trees and randomly samples them in order to obtain a majority “consensus” [22]. This random sampling helps to correct decision trees’ issue of overfitting to their training set. A decision tree is a tree-like model of predicting a target value based off observations about the target, where the observations are represented in the possible branches of the tree and the predictions are represented by the leaves [27].

Random forest classifiers have several notable advantages, including their accuracy with even a limited set of training data, low computational cost of training, and their good performance on non-linear problems. However, there are drawbacks, including the decreasing efficacy of adding additional training data (in contrast to a DNN, where more data almost always improves the accuracy), and overfitting is still a possibility, even with the random sampling of the decision trees [28].

3.2.3 Support Vector Machines

A support vector machine is a type of supervised learning model designed specifically for binary classification problems (classifying records into one of two subgroups) [23]. The SVM algorithm builds a model that represents the training data records as points in space, and attempts to plot them in such a way that a clear gap exists between the two subgroups. Then, when new records are received, a prediction is made by plotting the new record and determining which subgroup it belongs to based on which side of the gap it falls [29]. SVMs are natively able to perform linear classification, as well as being able to perform non-linear classification using the kernel method, where they implicitly map their inputs to high-dimensional feature spaces [30].

There are several advantages of SVM classification, most prominently being the ability to apply expert knowledge to the engineering of the kernel and achieve high accuracy [31]. However, this can also be a downside, as modifying the kernel can easily lead to over or underfitting [32]. In addition, SVM classification suffers from the same long training times as other deep learning methods.

3.3 Neural Network Architecture

Once the dataset is selected, the next step in the design process is architecting the structure of the neural network used for training and evaluating the data. Because neural networks are heuristic by nature, there is no single straightforward answer on which architecture will work best – it requires experimentation and refinement. However, there are several basic design decisions that have to be made prior to the implementation of the network.

The first high-level decision is whether or not the network should be recurrent (information traveling in both directions through the network) or feed-forward (information only traveling from input to output). Recurrent networks are primarily used when the network needs to retain information about what it has learned in the past, which is not a concern in this implementation [16]. Furthermore, the use of feed-forward networks allows for the use of back propagation error checking, a process where the network computes the error at the output layer and distributes it backwards through the network in order to correct the layer weights [17]. Feed-forward networks are considered generally acceptable for this type of classification problem, so this design is used in the network implementation.

The second decision concerns the number of hidden layers. All neural networks contain at least two layers: one input and one output. The defining feature of a deep neural network (the type selected for this thesis) is the existence of additional layers between the input and output, known as hidden layers [18]. The number and structure of these layers is often a matter of experimentation rather than design, but the current consensus is that the vast majority of problems obtain sufficient performance with a single hidden layer, and adding more layers only increases the potential for overfitting (an error where the model fits too closely to the set of training data and fails to accurately model the real word scenario) [19]. Therefore, the decision is made to start with a single hidden layer and refine the model as necessary.

Once this basic structure is determined, an exact number of nodes and an activation function needed to be selected for each layer. Although again this number is open to experimentation, there are a few commonly accepted guidelines for choosing the correct number of nodes. Generally, the size of the hidden layer should be between the size of the input and output layers (those sizes are determined by the data set, which will be discussed more in section 4). Additionally, the number of hidden nodes should be around 2/3 the size of the input layer, plus the size of the output layer. Finally, the number of hidden neurons should be less than twice the size of the input layer [20]. All of these guidelines are closer to suggestions than rules, but they provide a convenient starting point for the trial-and-error process that is building an accurate deep neural network. For the input layer, the number of nodes is equivalent to the number of features in the training data, 6 in this case. For the output layer, only a single node is needed for a binary classification problem. For the hidden layer, the decision is made to use 4 modes, which could then be refined through experimentation.

Finally, an activation function needs to be chosen for each layer. This is a function that calculates the weights of each node and edge after each training epoch, and is another facet of the DNN that can be adjusted. For a non-linear binary classification problem such as this, the rectified linear unit function (ReLU) is considered the appropriate choice [34]. In addition, a loss function (function to calculate the loss of the model after each epoch) and accuracy metric (metric for calculating the validation accuracy of each epoch) have to be chosen; for a binary classification problem, the user guide for the framework used in the implementation (discussed further in section 4.2) suggests the ‘binary_crossentropy’ and ‘binary_accuracy’ functions, respectively [36].

4 Implementation

4.1 Data Processing

The first step in implementing the chosen models is collecting the data from the two sources and curating it into a unified set of training data. The process used to collect the data from each source is described in the two following subsections.

4.1.1 National Vulnerability Database Parsing

For the NVD, data collection is rather straightforward: the NVD provides the data feeds from each year in the form of downloadable XML files, which contain the information on all discovered vulnerabilities for that year. In order to collect this data, the XML files from the years 2012-2017 were downloaded, and the relevant fields were extracted from them using BeaultifulSoup, a Python library for parsing HTML and XML files. Once these fields were parsed out for a given record, they were stored locally while the CVE reference table was checked (discussed in the next subsection). Because the values for all of these features (except the CVSS score) were a string, a simple encoding was implemented to convert these string values to integers (shown in *Table 1*).

Feature	**String**	**Integer**
Access Vector	Network	1
	Adjacent	2
	Local	3
	Physical	4
Complexity	Low	1
	High	2
	None	1
	Low	2
	High	3
Integrity	None	1
	Low	2
	High	3
Availability	None	1
	Low	2
	High	3

Table 1. String to integer feature encoding.

4.1.2 CVE Reference Table

In order to apply the labels to the training data, the CVE reference map discussed in subsection 3.1.3 was downloaded (using Requests) and parsed (using BeautifulSoup). This reference table provided a method for determining whether or not a given vulnerability from the NVD data feeds had been used in a publicly-known exploit. Once a record was parsed from the NVD file, the CVE ID number was extracted and checked against the reference table. If a match was found (indicating the existence of an exploit using that vulnerability), the label 1 was appended to the list of features for the current record. Otherwise, the label 0 was appended, indicating the lack of an existing exploit. Then, this list of features (including the binary label) was written to a CSV file, and the process was started over for the next record.

4.2 Model Creation

The next step in the implementation involves creating the various models that would be tested. The libraries, parameters, and structure of these models are discussed in the following subsections. Unless otherwise specified, Python was used in the implementation of these models.

4.2.1 Deep Neural Network

For the deep neural network, the library Keras was chosen for its simplicity and ease of use. Keras is a wrapper around the commonly-used Tensorflow library, a powerful open-source machine learning library published by Google. Keras abstracts many of the lower-level aspects of creating a neural network, and breaks the construction down into a few simple steps. Keras contains a set of pre-built model classes, and the Sequential model class was selected for this thesis, as it models a linear stack of layers, a very traditional DNN implementation. Once a Sequential model was instantiated, layers can be added individually to flesh out a full implementation.

As mentioned in the design section, the DNN used in this thesis has three main layers: an input layer, a single hidden layer, and an output layer. In addition, dropout layers were added before and after the hidden layer. A dropout layer is a specialized layer used to prevent overfitting of the model, which it does by deactivating a set percentage of randomly selected nodes during each training epoch. This forces the network to use different nodes on each epoch, and reduces the likelihood of the network simply “memorizing” the training data and overfitting [33].

4.2.2 Random Forest Classifier

Creating a random forest classifier is much less involved than the deep neural network. In order to create the classifier, the ‘scikit-learn’ library was used, which provides support for a variety of machine learning models, as well as functions for data processing and analysis. The only decision that had to be made was the number of estimators: generally, more is better, but performance can become asymptotic (reach a point where adding more decision trees results in only a marginal increase in accuracy) and computationally intensive [27]. For this case, 1000 decision trees were selected, with more to be added if performance was not satisfactory.

4.2.3 Support Vector Machine

Creating a support vector machine is similarly much simpler than a DNN, thanks to the ready-made model available from scikit-learn. The model accepts several parameters, but for this problem, the default values were deemed appropriate and used.

4.3 Training

To train the models, the first step was loading the data from the CSV file in which it was stored, which was accomplished using the data processing library Pandas. Pandas was also used to split the data into the feature set and corresponding labels, and a function from scikit-learn was used to split the data into training and testing data (80% training, 20% testing). There were approximately 50,000 total records (ranging from 2012 to 2017), leaving approximately 40,000 records for training and the remaining 10,000 for testing. This training data was then fed into the three models, each of which have a function that accepts a set of training data and returns a fitted model. In the case of the DNN, several additional parameters are required with the training function: number of epochs (iterations of the training and validation process), batch size (number of samples propagated through the network), and validation split (percentage of training data to hold back for validation after each epoch). These parameters are also subject to tuning as experimentation continues, but as a starting point, a set of 200 epochs, a batch size of 128, and a validation split of 20% were chosen.

4.4 Evaluation

The fitted models were then passed to another function for evaluation. This was also a very simple operation. The libraries in use require only a single function call (with the testing data and labels as parameters) in order to generate an accuracy score. Those model-specific evaluation functions were called inside another function, which repeats the evaluation process a user-specified number of times and returns an average of the scores. These are the scores that are discussed in detail in the next section.

5 Results

After the models were trained and ready to be evaluated, several different tests were run to compare performance. The first set of tests was a simple comparison of the three different models, using the starting parameters and an average of ten different evaluation scores. Once those tests were completed, several refinements were made to the models, and the new scores were recorded. After that, a single model was selected for further experimentation, including dynamic training of the model and refinement of the features used in training.

5.1 Algorithm Comparison

As previously mentioned, the first experiment involved evaluating each model ten times using the starting parameters discussed in the Implementation section. The averages of those evaluation scores are reported in *Table 2*.

Model	**Avg. Score**
Deep Neural Network	0.9405
Random Forest Classifier	0.1091
Support Vector Machine	0.9342

Table 2. Average of ten accuracy scores for each of the three implemented models.

The scores are given as decimals between 0 and 1, representing the percentage of accurate predictions of the testing data. From this data, it is immediately obvious that the deep neural network and the support vector machine vastly outperform the random forest classifier in accuracy. Although several refinements can be made to the random forest model to improve accuracy, which are discussed in the next subsection, it is clear that it is not well suited for this problem.

As previously mentioned, a single model was to be selected to use in the remainder of the experiments. Although the DNN and SVM both performed similarly, the SVM was selected for use in the remainder of the thesis, as the much higher computation time required for training the DNN (on the order of several hours, rather than minutes for the SVM) made it impractical to train multiple times on the various data sets. Therefore, the support vector machine was the model selected for use in the rest of the experimentation.

5.1.1 Model Refinements

Although all three models have room for improvement, tuning the parameters of the DNN and SVM models was left as potential future work, because the current accuracy was acceptable for experimentation. However, an attempt was made to improve the accuracy of the random forest model, namely by increasing the number of decision trees used. As mentioned previously, adding more decision trees generally results in higher accuracy, although it can become asymptotic (a point is reached where adding more trees results in only marginal increase in accuracy) and can vastly increase computation time.

For experimentation, the number of trees was increased several times, with the number of trees and accuracy being reported in *Table 3*.

Number of Trees	Avg. Score
5000	0.1263
10000	0.1432
25000	0.1372
50000	0.2729

Table 3. Accuracy of random forest models with increasing number of decision trees.

Although the accuracy does increase with the greater number of trees, it still falls well short of the accuracy displayed by the other two models. This reinforced the decision to select the SVM as the model with which to continue experimentation.

5.2 Dynamic Model Training

The next set of experiments involved dynamic training of the model: using a rolling window of training data. In this case, a window of one year was selected. The experiment was conducted by training the model only on the vulnerabilities from a given year, then evaluating the trained model against the vulnerabilities from the following year. This is in contrast to the typical static method of randomly selecting training and testing data from the same set of records. This method allows the model to be evaluated in a situation that models the real world: if the model was trained on a single recent year of data, how accurately would it perform on the next years' vulnerabilities? The graph below (*Figure 1*) shows the accuracy of the SVM when trained on a single year of data and evaluated on the following year.

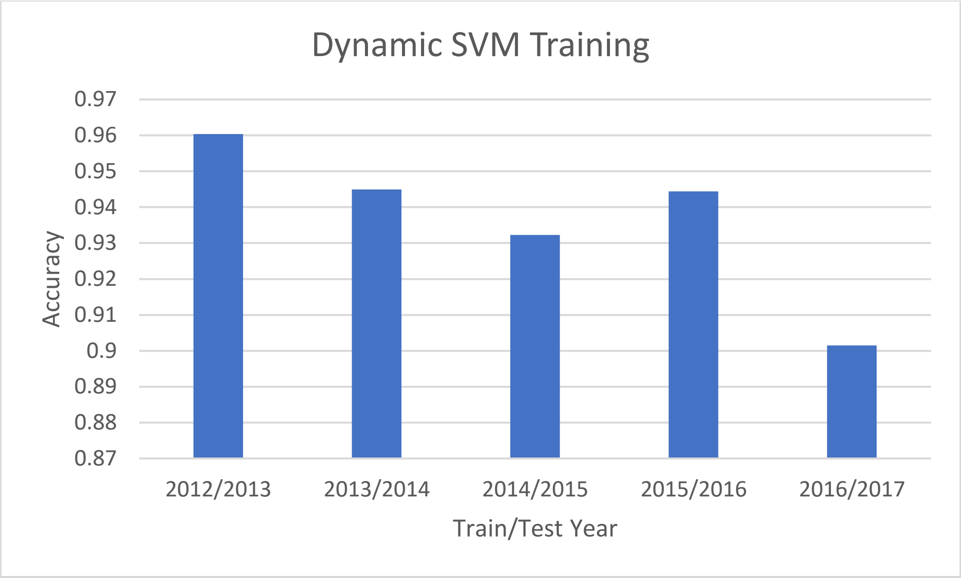

Figure 1. Accuracy of SVM when trained on single year and evaluated against the following year.

Although the model trained on 2015 data is an exception, the general trend is that the models become less accurate as the training data becomes more recent. This trend is somewhat expected, as fewer exploits have been discovered and catalogued for the more recent vulnerabilities. This makes the training data from recent years less useful, as it provides fewer examples of what an exploitable vulnerability looks like. However, even with the accuracy diminishing from year to year, the model still performed at greater than 90% accuracy for all tested years. This indicates that this model has the potential to be a useful tool for predicting the exploitability of new vulnerabilities.

5.3 Feature Comparison

The next set of experiments involved filtering the training data by certain features and comparing the accuracy. For each of the five discrete features (CVSS score was not considered, as its continuous nature is not well suited for filtering), the SVM model was retrained on only data matching each possible value, and then the average of ten scores was computed again and recorded. This was repeated with each of the five features, so that a correlation of features values and accuracy could be determined. The results are plotted in *Figures 2-6*.

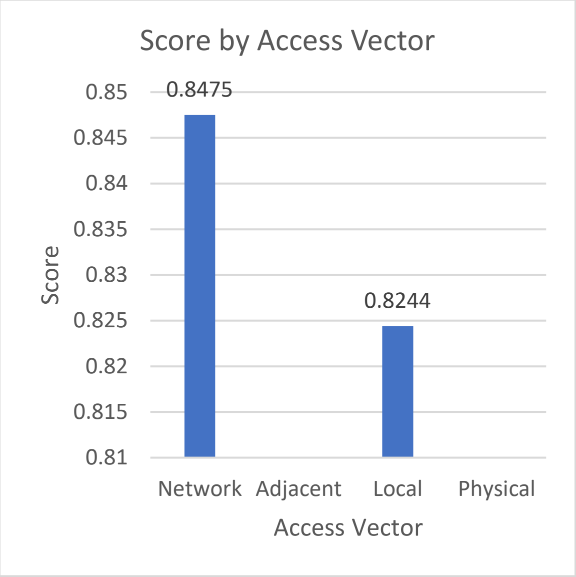

Figure 2. Effect of access vector on accuracy score. Note: no training data collected for Adjacent and Physical access vectors.

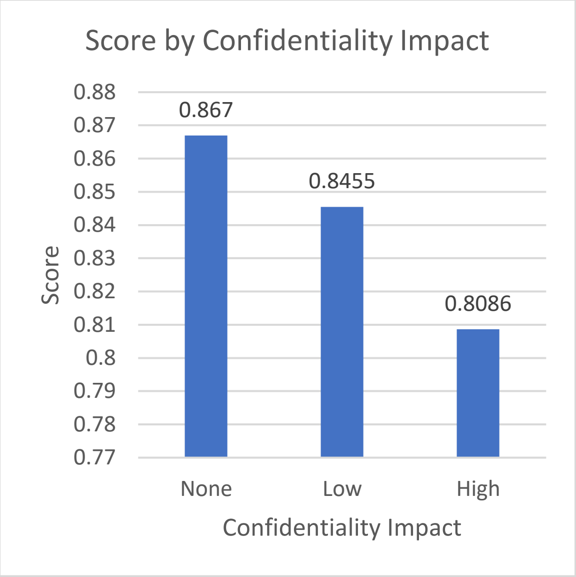

Figure 3. Effect of confidentiality impact on accuracy score.

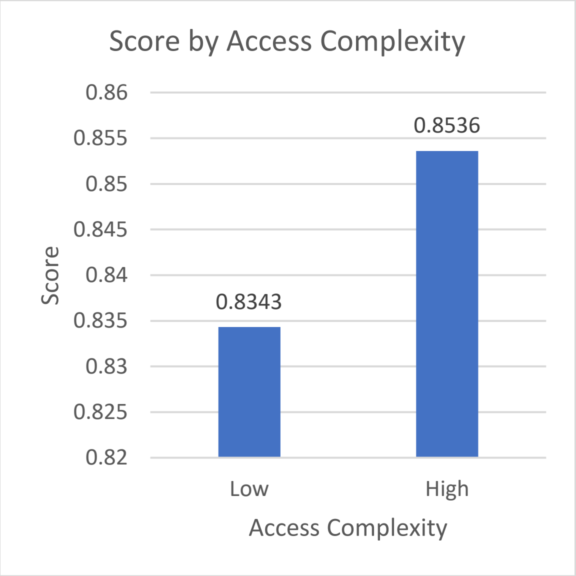

Figure 4. Effect of access complexity on accuracy score.

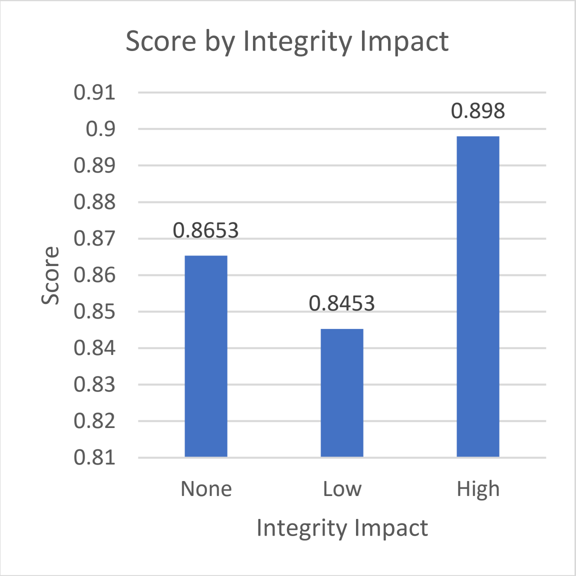

Figure 5. Effect of integrity impact on accuracy score.

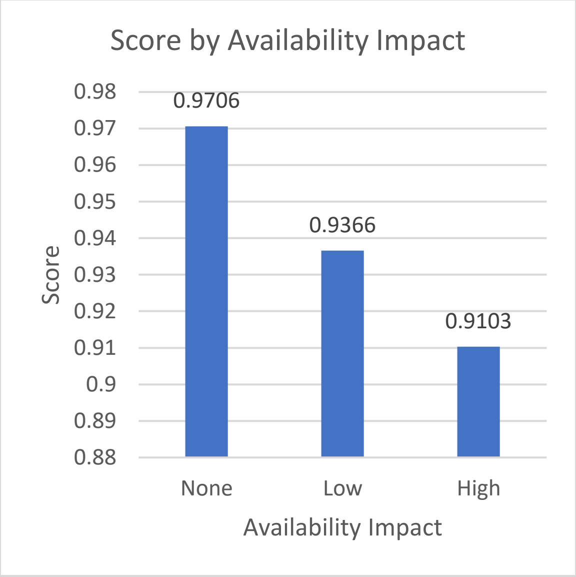

Figure 6. Effect of availability impact on accuracy score.

The above graphs demonstrate the efficacy of training the SVM model on the various features. Filtering by the various feature values does not appear to have a large impact on the accuracy, as the average difference between each potential value is only 1.6%. It does appear that filtering by availability impact increases accuracy, but more training data is required to confirm.

6 Conclusion

6.1 Summary

The purpose of this thesis was to explore methods for creating a learning-based tool to predict the exploitability of a given vulnerability based on its features. Several different machine learning models were implemented in order to test their effectiveness, including a deep neural network (DNN), a random forest classifier, and a support vector machine (SVM). In order to build a set of training data for the models, information was drawn from both the National Vulnerability Database and the Offensive Security Exploit Database. Once this information was collected and formatted, features were chosen to provide the most accurate information to the models. The chosen features included information such as the access vector of the vulnerability, the access complexity, and the CVSS score. In conjunction with these features, a CVE reference table was used to match known vulnerabilities with known exploits, which provided the target labels for the training data (binary labels – exploit or not an exploit). These features and labels were then used to train and test the various models, which were then compared for accuracy.

Out of the three chosen models, the deep neural network and support vector machine showed the highest accuracy (~ 93%). Refinements were made to the random forest model, which served to improve the accuracy, but it still fell well short of the other two models. Although the DNN showed a slightly higher accuracy (an average of 10 trials), the much longer time required for training a neural network led to the selection of the SVM to continue experimentation. Several more tests were conducted, including dynamic training of the model (using a single year of data to train, then testing the model on data from the following year) and filtering the training data by the various features. Several trends were discovered through these tests, while the accuracy remained high throughout (~ 80% - 90%). This suggests that this model could be developed into a usable tool that provides system administrators and other security professionals valuable insight into the exploitability of their systems.

6.2 Future Work

Although this work establishes a solid foundation, there are several improvements that can be made. The simplest improvement involves collecting more training data. The script used to parse the data feeds used in this thesis is easily extensible to cover a longer time period (and more records), but computation time was one limiting factor in choosing to restrict the data to the previous five years. Given more time, more records could be added to the training data, which has the potential to further boost the accuracy of all models. This additional data would specifically benefit the training of the DNN, which has the potential to become more accurate than the SVM and should certainly be considered for further experimentation. Additionally, because the SVM was selected for the more extensive testing in this thesis, it is possible that tuning the hyperparameters of the DNN (such as the learning rate, optimizer parameters, dropout rate, and activation function) could produce higher accuracy, even without additional training data.

After more data points are added to the training data, the next step should be to seek out additional sources of data. The current data sources only provide six features for use in training data, and it is very possible that building a more comprehensive feature set could boost the accuracy further. Other open source databases should be considered, and information such as the software vendor, required privileges, and any other potentially relevant features should be considered.

Finally, it could be useful for future researchers to update the data collection script to gather information about the timeline of the exploit: how long after the vulnerability was published did the exploit appear? Currently, the model provides no information about when a given vulnerability is likely to be exploited, only that an exploit is likely to be found. If this timeline information was collected from the NVD and Exploit-DB, this would allow researchers to consider time between publication of the vulnerability and exploit as a feature, and provide more detailed information about when an exploit could be expected to appear.

Once these further steps are taken, these rudimentary models can be implemented in highly advanced tools that provide the information needed for computer security professionals to protect their systems and help ensure the security of our digital world.

Acknowledgment

This thesis was advised by University of Arkansas professor Dr. Qinghua Li, and received valuable comments from professors Dr. Matt Patitz and Dr. Dale Thompson. This material is based upon work supported by the Department of Energy under Award Number DE-OE0000779.

References

[1] Symantec. “Internet Threat Security Report.” Mountain View, CA: Symantec Corporation, 2017. Accessed October 19, 2018. https://www.symantec.com/security-center/threat-report.

[2] Marco, B.; Nelson, B.; Joseph, A.D.; Tygar, J.D. (2010) "The Security of Machine Learning." Machine Learning 81, no. 2: 121-148. http://0-search.proquest.com.library.uark.edu/docview/750137890?accountid=836

[3] Ehrenfeld, Jesse M. "WannaCry, Cybersecurity and Health Information Technology: A Time to Act." Journal of Medical Systems 41, no. 7 (07, 2017): 1. doi:http://0-dx.doi.org.library.uark.edu/10.1007/s10916-017-0752-1. http://0-search.proquest.com.library.uark.edu/docview/1902028828?accountid=8361.

[4] Microsoft. “Microsoft Security Bulletin MS17-010: Critical” https://docs.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010

[5] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. (2017). Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. ACM Comput. Surv. 50, 4, Article 56 (August 2017). https://doi.org/10.1145/3092566

[6] Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. ACM Comput. Surveys (CSUR) 44, 3 (2012), 116]

[7] Brad Arkin, Scott Stender, and Gary McGraw. 2005. Software penetration testing. IEEE Security & Privacy 3, 1 (2005), 84–87.

[8] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing. Queue 10, 1 (2012), 20

[9] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. (2008). Using static analysis to find bugs. IEEE Softw. 25, 5 (2008), 22–29.

[10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. Commun. ACM (CACM) 53, 2 (2010), 66–75.

[11] Arthur L. Samuel. (1959). Some studies in machine learning using the game of checkers. IBM J. Res. Dev. 44, 1, 210–229

[12] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. Data Mining: Concepts and Techniques (3rd ed.). Morgan Kaufmann. Sean Heelan. 2011. Vulnerability detection systems: Think cyborg, not robot. IEEE Secur. Privacy 9, 3 (2011), 74–77.

[13] National Vulnerability Database. “National Vulnerability Database” Information Technology Laboratory, National Institute of Standards and Technology. https://nvd.nist.gov

[14] Offensive Security. “About the Exploit Database” https://exploit-db.com/about-exploit-db/

[15] National Vulnerability Database. “Common Vulnerabilities and Exposures” Information Technology Laboratory, National Institute of Standards and Technology. https://cve.mitre.org/index.html

[16] Mikolov, T.; Karafiát, M.; Burget, L.; Černocký, J.; Khudanpur, S. (2010). "Recurrent neural network based language model", In INTERSPEECH-2010, 1045-1048.

[17] Goh, A.T.C. (1995). “Back-propagation neural networks for modeling complex systems”. Artificial Intelligence in Engineering. 9 (3): 143-151. doi: 10.1016/0954-1810(94)00011-S

[18] Deng, L.; Hinton, G.; Kingsbury, B. (2013). “New types of deep neural network learning for speech recognition and related applications: an overview”. 2013 IEEE International Conference on Acoustics, Speech and Signal Processing. doi: 10.1109/ICASSP.2013.6639344

[19] Hawking, D.M. (2004). “The Problem of Overfitting”. Journal of Chemical Information and Computer Sciences. 44 (1): 1-12. doi: 10.1021/ci0342472

[20] Cao, W.; Mirchandani, G. (1989) “On hidden nodes for neural nets”. IEEE Transactions on Circuits and Systems. 36 (5): 661-664. doi: 10.1109/31.31313

[21] Deng, L.; Yu, D. (2014). "Deep Learning: Methods and Applications". Foundations and Trends in Signal Processing. 7 (3–4): 1–199. doi:10.1561/2000000039.

[22] Ho, Tin Kam (1995). Random Decision Forests. Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14–16 August 1995. pp. 278–282. Archived from the original (PDF) on 17 April 2016. Retrieved 5 June 2016.

[23] Cortes, Corinna; Vapnik, Vladimir N. (1995). "Support-vector networks". Machine Learning. 20 (3): 273–297. doi:10.1007/BF00994018.

[24] Alpaydin, Ethem (2010). Introduction to Machine Learning. MIT Press. p. 9. ISBN 978-0-262-01243-0.

[25] Bengio, Yoshua (2009). "Learning Deep Architectures for AI". Foundations and Trends in Machine Learning. 2 (1): 1–127. doi:10.1561/2200000006.

[26] Hinton, G. E. (2010). "A Practical Guide to Training Restricted Boltzmann Machines". Tech. Rep. UTML TR 2010-003.

[27] Rokach, Lior; Maimon, O. (2008). Data mining with decision trees: theory and applications. World Scientific Pub Co Inc. ISBN 978-9812771711.

[28] Hastie, Trevor; Tibshirani, Robert; Friedman, Jerome (2008). The Elements of Statistical Learning (2nd ed.). Springer. ISBN 0-387-95284-5.

[29] Ben-Hur, Asa; Horn, David; Siegelmann, Hava; and Vapnik, Vladimir N.; "Support vector clustering"; (2001); Journal of Machine Learning Research, 2: 125–137

[30] Hsu, Chih-Wei; Chang, Chih-Chung & Lin, Chih-Jen (2003). A Practical Guide to Support Vector Classification. Department of Computer Science and Information Engineering, National Taiwan University. Archived (PDF) from the original on 2013-06-25.

[31] Crammer, Koby & Singer, Yoram (2001). "On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines". Journal of Machine Learning Research. 2: 265–29

[32] G. C. Cawley and N. L. C. Talbot (2010). “Over-fitting in model selection and subsequent selection bias in performance evaluation”. Journal of Machine Learning Research. 11: 2079-2107

[33] Hinton, Geoffrey E.; Srivastava, Nitish; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan R. (2012). "Improving neural networks by preventing co-adaptation of feature detectors". arXiv:1207.0580

[34] Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011). Deep sparse rectifier neural networks. AISTATS.

[36] Keras Documentation. “Metrics”. https://keras.io/metrics/