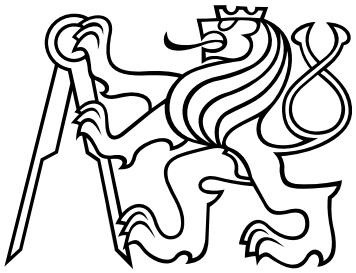
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of electrical engineering



BACHELOR THESIS

2016 **Karel GAVENČIAK**

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Karel G a v e n č i a k

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Learn and Predict Metasploit Exploit Ranks from Available Vulnerability Information

Guidelines:

Metasploit tool contains an interesting exploit feature called rank, which describes reliability and/or success probability of the given exploit. Attackers often order exploits in descending rank values, since exploits with high rank are most reliable, likely to succeed and least detectable. However, there are many vulnerabilities in National Vulnerability Database (NVD), which do not have exploits in Metasploit (yet). Learning their ranks as soon as possible can direct the network administrator to focus on important and likely exploited vulnerabilities in his network.

Use AI approaches (machine learning, data mining, etc.) to predict the rank of vulnerabilities based on the available data (e.g., CVE specification, CVSS vectors, etc.) as follows:

1. Study the Metasploit's available information about exploits, such as ranks, etc.
2. Study available information about vulnerabilities.
3. Select principal features of vulnerabilities for learning the rank.
4. Propose an AI approach for learning the exploit's rank of vulnerability.
5. Implement algorithm(s) and evaluate their precision using predictive analytics.

Bibliography/Sources:

[1] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: Learning to classify vulnerabilities and predict exploits," in Proc. of 16th Int. Conf. on Knowledge discovery and data mining, 2010, pp. 105-114.

[2] Mell, Peter, Karen Ann Kent, and Sasha Romanosky. The common vulnerability scoring system (CVSS) and its applicability to federal agency systems. US Department of Commerce, National Institute of Standards and Technology, 2007.

[3] Hartanto, Teddy. "Penetration Testing: Testing the Security of Computer Systems." CS2107-Semester IV 2014-2015: 65.

Bachelor Project Supervisor: Ing. Karel Durkota

Valid until: the end of the summer semester of academic year 2016/2017

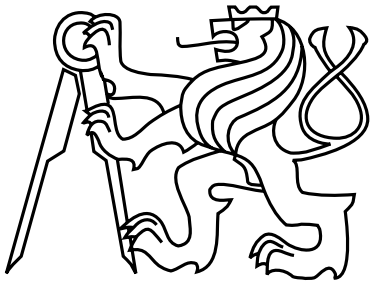
L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, December 11, 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics



Bachelor's Project

Learn and Predict Metasploit Exploit Rank from Available Vulnerability Information

Karel Gavenčiak

Supervisor: Ing. Karel Durkota

Study Programme: Open Informatics

Field of Study: Computer and Information Science

May 26, 2016

Aknowledgements

I would like to express my sincere gratitude to my advisor Ing. Karel Durkota for his time, advice and feedback that he dedicated to my project and for creating friendly working environment. Also, I would like to thank my parents for their support during my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 25. 5. 2016 .....

Abstract

Vulnerabilities prioritization is an important part of security administrators' work. The aim of this work is to focus on estimating the risk that each vulnerability presents by predicting rank of its exploit in Metasploit database. As a base for representing vulnerabilities, we created multidimensional vectors from publicly available information. For predicting we used Support Vector Machine and Random Forest algorithms with several normalization methods. With this approach, we were able to predict the existence of Metasploit exploit with 97% accuracy and its rank with 44% accuracy. Results of this project prove that machine learning can be useful in prioritizing vulnerabilities.

Abstrakt

Prioritizace vulnerabilit je důležitou součástí práce systémových bezpečnostních administrátorů. Cílem této práce je, zaměřit se na odhadnutí risku, jaký každá vulnerabilita představuje pomocí předpovídání jejího příslušného exploit ranku v databázi Metasploit. Z veřejně dostupných dat jsme vytvořili mnoharozměrné vektory, které slouží jako základ pro reprezentaci vulnerabilit. Pro předpovídání ranku jsme využili algoritmy Support Vector Machine a Random Forest s několika různými normalizačními techinkami. Ve výsledku jsme byli schopni předpovědět existenci exploitu s úspěšností 97% a rank exploitu s úspěšností 44%. Výsledky ukazují, že použití strojového učení může být velmi užitečné při určování priorit pro systémové vulnerability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rising number of internet connected devices in recent years makes the demands for computer security higher. More used devices present possible entrance points to targeted networks, thus, it is becoming easier to attack targeted systems.

The common process of computer attack is that an adversary finds weak spots in targeted network and attacks them with an exploit – a set of methods and procedures that take advantage of vulnerabilities in the system. When the exploit is successful, the attacker is given root privileges to use the system as its administrator. The whole process is even easier as most of these exploits are often already predefined and publicly available online.

Hence, system security administrators are often facing difficulties when encountering multiple vulnerabilities in their systems. They usually work with limited resources and cannot fix all flaws at once. That is why vulnerabilities need to be prioritized with aspect to their risks. The problem is that it requires a deep analysis of each vulnerability individually which is inefficient and time-consuming.

1.1 Thesis Assignment

The aim of our work is to create a tool that predicts the existence and quality of exploit for particular vulnerabilities. This tool should offer a way how to estimate the possible risk that vulnerabilities present and prioritize them when securing the system.

In our approach, we use Support Vector Machines and Random Forest algorithms to predict Metasploit ranking for vulnerability exploits. For representing each vulnerability we create vector consisting of selected features which are obtained from publicly available information. Further, we evaluate the performance of used classifiers and examine the influence of several different normalization methods and use of language processing.

1.2 Thesis Structure

In Chapter 2 we take a closer look on sources of publicly available vulnerability information. We describe Metasploit and vulnerability and exploit datasets. Chapter 3 describes features extraction and their analysis. Chapter 4 provides theoretical background about

learning algorithms that we used in our project. Chapter 5 overviews the experiments,their results. Chapter 6 is a discussion and analysis of our results. Last, Chapter 7 is the conclusion of our project.

1.3 Related Work

Demands for quick and accurate vulnerability prioritization tools are large. These tools are getting, even more, importance because the time available to patch vulnerable system is shrinking. Eschelbeck [9] showed the exploitation cycle is shortening (i.e. the time from disclosure of a vulnerability and exploit availability) and growing trend of zero-day exploits (exploits which are available at the time of disclosure of a vulnerability) was also proved by Frei [11].

Most known standardized methodologies for risk estimation is Common Vulnerabilities Scoring System (CVSS). Although CVSS is widely spread and recognized technique how to prioritize vulnerabilities, even when constructed thoroughly by experts, values assigned to the vulnerabilities are still from small range.

Problems about deriving risk from CVSS score were proved by Allodi and Massacci [3]. In their research, they have focused on exploits that appear in the wild, i.e. exploits that were really used by attackers. By comparing vulnerabilities from National Vulnerability Database, their CVSS score and existing exploits (from databases and also black market) they provide assessment whether it is possible to estimate real risk for vulnerabilities by their CVSS score. As a result mentioned approach did not prove to be an efficient indicator that exploit for a particular vulnerability will appear in the wild.

Caution when using CVSS score for prioritizing vulnerabilities is also recommended by Rieke in [16]. He mentioned that a vulnerable software itself does not necessarily imply that someone can exploit a vulnerability. Other requirements are for running the vulnerable software and also if the targeted system is reachable on the port that vulnerable software is using.

Frühwirth and Männistö [12] advise to improve CVSS with metrics related to each enterprise individually. Results show that it is worth not only because improving quality aspect but also in reference to the cost of additional effort.

Bozorghi et al. showed a different approach to the problem of estimating vulnerability risk [5]. Instead of classifying vulnerability with a score based on expert's analysis, authors are using machine learning (ML) principles. They labeled vulnerabilities as positive or negative depending whether there is an existing exploit on them. With the use of the bag-of-words algorithm, they extracted high-dimensional vector with a total number of 93578 features for each vulnerability. They use SVM for predicting whether there is an existing exploit for a vulnerability or how soon it might be created. Results of their work show that it is possible to predict the existence of exploit with 89.8% accuracy.

A similar approach was presented by Edkrantz et al. [8]. In addition, authors predict the existence of exploit using not only SVM but also other ML approaches as Random forest and k-Nearest Neighbors. The accuracy of their prediction is slightly less than Bozorgi's, about 83%. Another use of ML in the field of computer security and risk estimation was by

Labuť [14]. He used it to estimate the cost which adversaries need to pay in order to obtain successful exploit codes.

ML and statistics methods seem to serve well as a method to prioritize vulnerabilities. Although accuracy of prediction exploit is high in previous researches, they focus only on predicting the existence of exploit but not on its severity. Previous works have also omitted one important aspect. Vulnerability description is transformed into multidimensional vectors by bag-of-words but there is no consideration of the negations occurring in those sentences.

In our approach, we try to predict not only the existence of exploit but its severity, by predicting exploit rank. In addition, when extracting the data, we are resolving negations in descriptions and distinguish their meaning from normal sentences.

Chapter 2

Vulnerability Information and Data Gathering

2.1 Data Sources

2.1.1 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures [1] is a dictionary of previously discovered vulnerabilities. Each vulnerability, when recorded gets its unique identifier (also referred as CVE name, CVE number, CVE-id), that helps to distinguish it among other imperfections and flaws. Multiple vendors on the market (e.g. Cisco, IBM, McAfee or Symantec corporation) have the CVE dictionary incorporated into their products, which allows quickly finding additional information about a solved problem when knowing this common identifier.

2.1.2 Common Vulnerability Scoring System

Common vulnerability scoring system (CVSS), currently with its second version, provides a standardized way to produce a numerical score for each vulnerability that characterizes its severity. This score is called total CVSS score and it is computed from three main groups, which are further divided into several different metrics characterizing the vulnerability (all groups and metrics are displayed in Figure 2.1). The Base group refers to main features of a vulnerability, Temporal presents characteristics that change during the time and Environmental are related to a specific enterprise or environment where the user is located. The most important of them is the Base group because it is the only necessary part to compute the total score. The two other groups only adjust the final score value and are not obligatory.

Access Vector, Access Complexity, and Authentication create together so-called Exploitability sub-score that should characterize how easily is the vulnerability exploitable. Metrics as Confidentiality, Integrity, and Availability create together Impact sub-score. It represents the risk which the vulnerability presents in a case of successful exploitation by an adversary. The total score is calculated from Exploitability and Impact sub-scores.

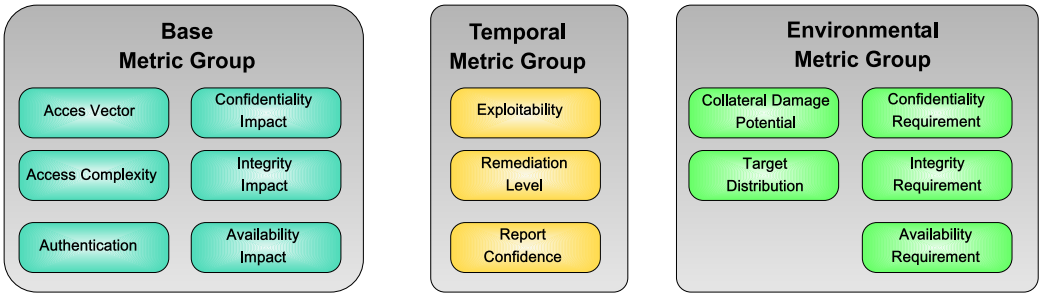


Figure 2.1: CVSS metric groups.

2.1.3 National Vulnerability Database

National Vulnerability Database (NVD) [2] is the repository of the vulnerability information, managed by the U.S. government. It includes databases of security checklists, security related software flaws, misconfigurations, product names, and impact metrics. All vulnerabilities discovered in the past are tracked in data feeds with their detailed description. Those feeds contain CVE-id of the vulnerability and detailed information like affected products, references to additional information, date of discovery, last modification date, CVSS metrics for vulnerability and vulnerability summary – a description of vulnerability created thoroughly by experts.

NVD has a good reputation because works under the auspices of state government and operates in computer security for a long time (it tracks vulnerabilities since 1999), hence we decided to use their information as the main source when extracting data for our project.

2.1.4 Metasploit

As was presented by Hartanto in [13], first created in 2003, Metasploit is a system security tool for penetration testing and revealing weak spots in networks. In our Thesis we work with Metasploit Framework, free open source version, accessible via command line, that provides the user almost the same possibilities as Metasploit except a few complicated tasks (e.g. credentials brute forcing or web app testing). The database that Metasploit contains counts up over 1300 exploits and 2000 modules it allows simulate real world attacks of different types. These exploits are labeled with a rank, referring to exploit's reliability. Ranks are as follows in Table 2.2. These labels allows us to distinguish the exploits and provide us a better knowledge about their possible behavior without actually studying them and offer us a way how to predict their final success.

2.2 Vulnerability Dataset

During the gathering of the data for learning, we were able to collect information about a total number of 64100 vulnerabilities from NVD. The total amount of vulnerabilities per

rank	description
excellent	The exploit will never crash the service. This is the case for SQL Injection, CMD execution, RFI, LFI, etc. No typical memory corruption exploits should be given this ranking unless there are extraordinary circumstances.
great	The exploit has a default target AND either auto-detects the appropriate target or uses an application-specific return address AFTER a version check.
good	The exploit has a default target and it is the "common case" for this type of software (English, Windows XP for a desktop app, 2003 for server, etc).
normal	The exploit is otherwise reliable, but depends on a specific version and can't (or doesn't) reliably autodetect.
average	The exploit is generally unreliable or difficult to exploit.
low	The exploit is nearly impossible to exploit (or under 50%) for common platforms.
manual	The exploit is unstable or difficult to exploit and is basically a DoS. This ranking is also used when the module has no use unless specifically configured by the user.

Table 2.2: Metasploit ranks

year is in Table 2.3. Figure 2.4 shows an average number of exploits per one vulnerability. From the frequency of gathered vulnerabilities we can see that even when the number of vulnerabilities per year is increasing only slightly, the amount of exploits per vulnerability grows faster. It indicates that raising number of devices connected to the internet nowadays probably brings more attention of adversaries. Lower values in the last two years may be either due to raising focus of software developers on system security or also due to the fact that more exploit codes can still be developed because software products released in particular time are still commonly used.

Year	# of vulnerabilities	# of exploits
2002	3523	30
2003	1465	19
2004	2671	45
2005	4685	66
2006	6997	81
2007	6510	88
2008	2500	43
2009	4887	97
2010	4849	107
2011	3587	97
2012	5092	121
2013	5604	148
2014	5676	84
2015	6051	92

Table 2.3: Vulnerabilities and exploit frequency in past years.

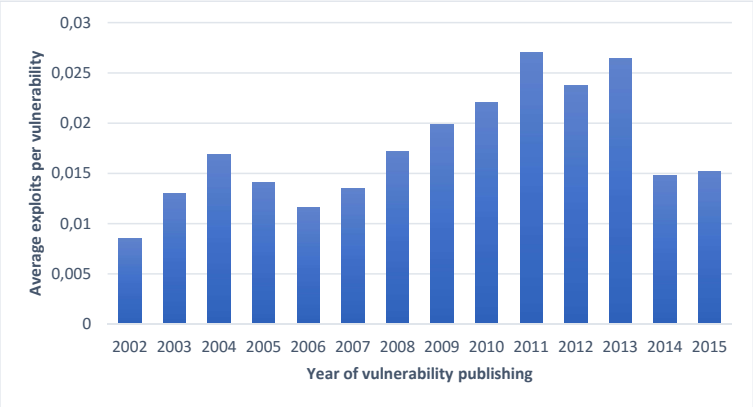


Figure 2.4: Average number of exploits per one vulnerability.

2.3 Exploit Dataset

Although we were able to examine a total number of 64100 vulnerabilities, existing exploit was reported only for 1123 of them. The number of exploits reported was lower even when Metasploit itself disposes with a total number of 3700 exploits and modules, only about a third of them is linked with CVE-id of a vulnerability that they are related to, which was necessary for our research.

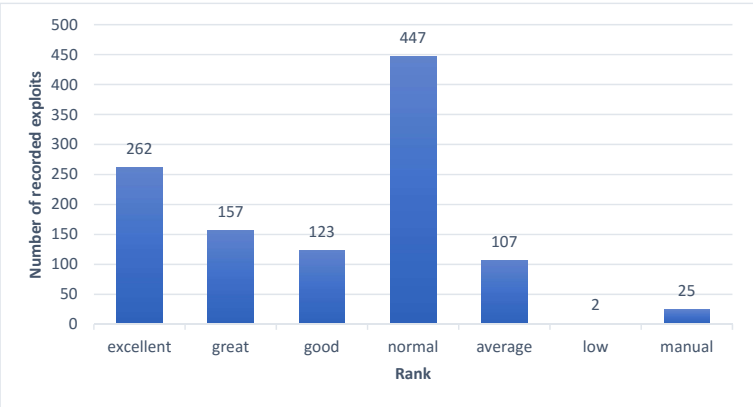


Figure 2.5: Frequency of ranks.

The number of exploits reported per rank is shown in Figure 2.5. The most frequent one shows to be *normal* rank, followed by *excellent*. Least frequent with its 2 appearances was *low* rank, exploits with this label might not be used as much due to their unreliability, thus, there is probably fewer of them in Metasploit database.

In final learning and predicting process, two exploits with low rank were not enough, so we have decided to integrate them to group labeled with average rank (those two ranks are closest to each other in terms of reliability). Final set of all classes from our dataset was $C = \{excellent,\ great,\ good,\ normal,\ average,\ manual,\ no\ exploit\}$.

Still all the data obtained from Metasploit would create a very unbalanced dataset for learning. Class *no exploit* would highly over-size the rest of the classes. Therefore, we decide to under–sample and use only 155 (an average amount of samples for one exploit rank class) samples from *no exploit* class and create dataset S with a total number $N = 1278$ of vulnerabilities for our experiments.

Chapter 3

Features Extraction and Analysis

3.1 Features Extraction

For predicting exploit rank we created set F, containing a total number of 4873 features. For each vulnerability $s_i \in S$ we created multidimensional vector $\mathbf{v}_i = (f_j^i | f_j \in F)$ where $f_j^i \in \mathbb{R}$ represent value of feature f_j for vulnerability s_i. The majority of features comes from the bag-of-words representation of vulnerability summaries. Although this inspection gave us large amount of features, we decided to add more important features.

Our final dataset for learning and experiments has a form $L = \{\mathbf{v}_i, c_i\},\ i = 1, \cdots, N$ where $c_i \in C$ is rank of exploit for vulnerability s_i.

In following sections, we demonstrate extraction of specific features using the example of a vulnerability record in Figure 3.1.

3.1.1 Affected Products

f_{prod} is a feature that represents the number of affected products by vulnerability s. It is another of features extracted from NVD (Figure 3.1 part 2). We expect vulnerabilities that affect a greater amount of products to be more attractive for attackers and exploit creators since they will be able to attack more systems with one exploit and thus have a bigger chance for a success. Therefore, we consider the exploit for vulnerability with a higher number of affected products to be more likely to occur.

3.1.2 Date Differences

Three added features come from inspiration of results of previous Bozorghi's work. In top ten negative influential features, four of them are based on vulnerability dates. From NVD we extracted last modification date and published date of vulnerability records (Figure 3.1 part 3). For each vulnerability, we constructed three features to characterize the vulnerability life cycle.

```
  <entry id="CVE-2013-2143">                                                    Part 1
    <vuln:vulnerable-configuration id="http://www.nist.gov/">
      <cpe-lang:logical-test operator="OR" negate="false">
        <cpe-lang:fact-ref name="cpe:/a:redhat:network_satellite:-"/>
        <cpe-lang:fact-ref name="cpe:/a:katello:katello:1.5.0-14"/>
      </cpe-lang:logical-test>
    </vuln:vulnerable-configuration>
    <vuln:vulnerable-software-list>                                             Part 2
      <vuln:product>cpe:/a:katello:katello:1.5.0-14</vuln:product>
      <vuln:product>cpe:/a:redhat:network_satellite:-</vuln:product>
    </vuln:vulnerable-software-list>
    <vuln:cve-id>CVE-2013-2143</vuln:cve-id>
    <vuln:published-datetime>2014-04-17T10:55:05.730-04:00</vuln:published-datetime>  Part 3
    <vuln:last-modified-datetime>2014-04-17T11:57:43.043-04:00</vuln:last-modified-datetime>
    <vuln:cvss>                                                                 Part 4
      <cvss:base_metrics>
        <cvss:score>6.5</cvss:score>
        <cvss:access-vector>NETWORK</cvss:access-vector>
        <cvss:access-complexity>LOW</cvss:access-complexity>
        <cvss:authentication>SINGLE_INSTANCE</cvss:authentication>
        <cvss:confidentiality-impact>PARTIAL</cvss:confidentiality-impact>
        <cvss:integrity-impact>PARTIAL</cvss:integrity-impact>
        <cvss:availability-impact>PARTIAL</cvss:availability-impact>
        <cvss:source>http://nvd.nist.gov</cvss:source>
        <cvss:generated-on-datetime>2014-04-17T11:57:42.980-04:00</cvss:generated-on-datetime>
      </cvss:base_metrics>
    </vuln:cvss>
    <vuln:cwe id="CWE-20"/>
    <vuln:references xml:lang="en" reference_type="UNKNOWN">                    Part 5
      <vuln:source>BID</vuln:source>
      <vuln:reference href="http://www.securityfocus.com/bid/66434"
xml:lang="en">66434</vuln:reference>
    </vuln:references>
    <vuln:references xml:lang="en" reference_type="UNKNOWN">
      <vuln:source>OSVDB</vuln:source>
      <vuln:reference href="http://www.osvdb.org/104981" xml:lang="en">104981</vuln:reference>
    </vuln:references>
    <vuln:references xml:lang="en" reference_type="UNKNOWN">
      <vuln:source>EXPLOIT-DB</vuln:source>
      <vuln:reference href="http://www.exploit-db.com/exploits/32515"
xml:lang="en">32515</vuln:reference>
    </vuln:references>
    <vuln:references xml:lang="en" reference_type="UNKNOWN">
      <vuln:source>MISC</vuln:source>
      <vuln:reference
href="http://packetstormsecurity.com/files/125866/Katello-Red-Hat-Satellite-users-update_roles-M
issing-Authorization.html"
xml:lang="en">http://packetstormsecurity.com/files/125866/Katello-Red-Hat-Satellite-users-update
_roles-Missing-Authorization.html</vuln:reference>
    </vuln:references>
    <vuln:summary>The users controller in Katello 1.5.0-14 and earlier, and Red Hat Satellite,
does not check authorization for the update_roles action, which allows remote authenticated
users to gain privileges by setting a user account to an administrator account.</vuln:summary>
  </entry>
                                                                                Part 6
```

Figure 3.1: Example of vulnerability record from NVD.

- $f_{dateSinceModified}$ – Difference of actual date and last modification date. Value represent time from the last update of vulnerability record.

- $f_{dateSincePublished}$ – Difference of actual date and published date. It represents the age of vulnerability. Exploit for older vulnerabilities is considered less likely.

- $f_{dateDiff}$ – Difference of last modification date and published date. The Larger value in difference means that the record was modified later after vulnerability disclosure, which probably means there were patches released from the vendor and the risk of vulnerability was mitigated.

3.1.3 CVSS Metrics

Because CVSS is criticized and not well recommended for prioritizing vulnerabilities, we decided to use all metrics available in NVD to create additional features. Features named $f_{cvss_{1-7}}$ present Total CVSS score, Access Vector, Access Complexity, Authentication, Confidentiality Impact, Integrity Impact, Availability Impact.

3.1.4 References

Feature named f_{ref} presents number of references and it is also extracted from NVD (Figure 3.1 part 5). These links refer to additional information about the vulnerability as whether it was recorded in any other database. Vulnerabilities that are more dangerous present a higher risk for system security thus they get more into community's awareness. We expect a higher value to indicate that there will be an existing exploit for the vulnerability.

3.1.5 Bag of Words representation

From NVD, we created set D of all vulnerability descriptions for vulnerabilities $s_i \in S$ (Figure 3.1 part 6). Then we cleared each description $d_i \in D$ from numbers, punctuation and stop words (*the*, *of*, etc.) and split them into set of words w_i. We created set of all words $W = \bigcup_{i=1}^{N} w_i$. Each word $w \in W$ we present as one feature f_w. Value f_w^i is equal to number of occurrences of w in particular description for vulnerability s_i.

Vulnerability shown in Figure 3.1 has the following summary:

The users controller in katello 1.5.0–14 and earlier, and Red Hat Satellite, does not check authorization for the update_ roles action, which allows remote authenticated users to gain privileges by setting a user account to an administrator account.

After cutting of stopwords and numbers we gain these features with their respective values:

$f_{users} = f_{account} = 2$,$f_{controller} = f_{katello} = f_{earlier} = \cdots = f_{administrator} = 1$.

The rest of the bag of words features for words that do not occur in this summary would have zero value.

3.2 Most Frequent Features

The frequency of additional features that comes from vulnerability characteristics like CVSS metrics, date based features, the number of affected products or references is shown in Table 3.2. These features are present in the database almost for every record.

Feature name	Frequency
$f_{dateSincePublished}$	1278
$f_{dateSinceModified}$	1277
$f_{cvssTotalScore}$	1273
$f_{cvssAccessVector}$	1273
$f_{cvssAccessComplexity}$	1273
$f_{cvssAutentication}$	1273
$f_{numberOfReferences}$	1265
$f_{numberOfProducts}$	1265
$f_{dateDiff}$	1226
$f_{cvssConfidentialityImpact}$	1131
$f_{cvssAvailabilityImpact}$	1108
$f_{cvssIntegrityImpact}$	1089
Total number of vectors	1278

Table 3.2: Frequency of additional features.

The most frequent words to appear in the vulnerability descriptions are shown in the Table 3.3. Most of them (words like *remote, attackers, allow*) are neutral in the meaning of exploit reliability. Although words buffer, overflow, stack, service, offer closer characteristic of exploit type, and might help in predicting rank.

Word	Frequency
remote	1147
attackers	1088
allows	1082
arbitrary	979
execute	870
code	731
buffer	463
vulnerability	397
overflow	384
service	302
Total number of vectors	1278

Table 3.3: 15 Most frequent features coming from Bag of Words representation.

3.3 Language Processing

As it was mention in the introduction, we base our work on Bozorghi's previous research [5]. In their work, there were no language processing methods which would take into consideration the negations in vulnerability descriptions. We found that an important part in creating the data features because negation in vulnerability summary can completely change the meaning of the sentence. When exploring vulnerabilities summaries in detail, we have found that from all negative expressions only the word *not* occurs there. These sentences represented 368 of the total amount. That is 16 percent of all vulnerabilities, surely not a negligible amount. *Not* is in these sentences always followed by the verb that it is connected to or by an adverb and a verb.

Therefore, in our approach we have in additionally implemented solution that finds negations in sentences, determines the verb connected to the negation by the part-of-speech-tagging method.

In case of record from shown in Figure 3.1, word *check* would be substituted with word *not_checked* and new feature f_{not_check} would be created to describe better the meaning of the sentence.

3.4 Normalization Methods

Normalization of data instances is widely recommended in machine learning. The main advantage is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges and it shortens the training time and improves the final accuracy. In our work we have used and analyzed results of three different methods. One was setting the maximum value for each feature equal to one, the next was setting the summation over all feature values to one and the third was the Term–Frequency–Inverse–Document–Frequency (TF–IDF) method, widely used for document representation.

- **Normalization by feature max value**:

$$\overline{v}_{i,j} = \frac{f_j^i}{\max_{s_k \in S} f_j^k}$$

  Where $\overline{v}_{i,j}$ is normalized value of f_j^i. The denominator represents a maximum number of feature $f_j \in F$ over all records in dataset S.

- **Normalization by feature summation**:

$$\overline{v}_{i,j} = \frac{f_j^i}{\sum_{s_k \in S} f_j^k}$$

  Where the value f_j^i is normalized by summation of value of feature $f_j \in F$ over all records.

- **TF-IDF normalization:**

$$\mathrm{tf}_{i,j} = \frac{f_j^i}{\sum_{s_k \in S} f_j^k}$$

tf means term frequency, in our case value of feature f_j^k for vulnerability $s_k \in S$, divided by summation of values of feature $f_j \in F$ over all document records in dataset S.

$$\mathrm{idf}_j = \log \frac{|D|}{|\{i|f_j^i > 0\}|}$$

idf means inverse document frequency. Where $|D|$ is total number of vulnerability descriptions and $|\{i|f_j^i > 0\}|$ is number of records for which the feature f_j is greater than zero. The final normalized value is:

$$\overline{v}_{i,j} = \mathrm{tf}_{i,j} + \mathrm{tf}_{i,j} \cdot \mathrm{idf}_j$$

Chapter 4

Technical Background

4.1 SVM

When classifying linearly separable binary labeled data, we are trying to find the hyperplane that separates the data feature space into two parts according to the classes of training samples. But often there is more of such hyper planes possible to be found. Support Vector Machine (SVM) is an algorithm that finds mentioned hyperplane in a way that it maximizes the margin between itself and the closest data points from each class.

The problem is that our data are not binary, we have seven different classes with respect to seven different Metasploit ranks. In this case, for using SVM we need to binarize the data, separately for each class and create seven different classifiers.

For creating such classifier for class $c \in C$ suppose we have our training data $\{\mathbf{v}_i, y_i\}$ where $i = 1, \cdots, N$ and $y_i = +1 \ \forall i : c_i = c,\ y_i = -1 \ \forall i : c_i \neq c$ and a hyperplane that separates the positive from the negative samples. Points which lie on the hyper plane satisfy $\mathbf{w} \cdot \mathbf{v} + b = 0$, where $\mathbf{w}$ is normal to the hyperplane $|b|/||\mathbf{w}||$ is the perpendicular distance from the hyperplane to the origin and $||\mathbf{w}||$ is the euclidean norm of $\mathbf{w}$.

In order to maximize the margin the task can be formulated as follows:

$$\min \frac{1}{2}||\mathbf{w}||^2$$

subject to

$$y_i(\mathbf{v}_i \cdot \mathbf{w} + b) - 1 \geq 0 \ \forall_i$$

After we create classifiers for each class, when new record is classified we give it label of the class whose classifier gives the highest value for the record.

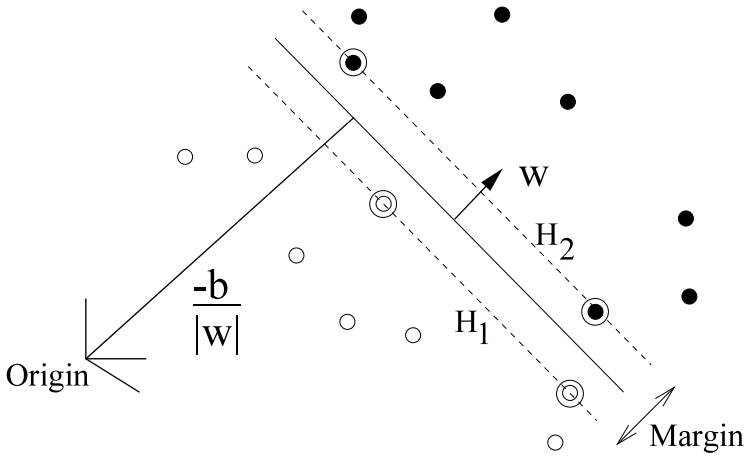


Figure 4.1: Hyperplane through two linearly separable classes [7]

4.1.1 Slack Variables

In real world, the data we work with are often noisy, and even when the measurements are taken precisely, some errors may occur. For this cases we can implement so called slack variables. Task to minimize is then:

$$\frac{1}{2}||w||^2 + C \cdot \sum_i \xi_i$$

subject to:

$$y_i(\mathbf{w} \cdot \mathbf{v}_i + b) \geq 1 - \xi_i, \text{ for } i = 1, \ldots, N$$
$$\xi_i \geq 0, \text{ for } i = 1, \ldots, N$$

Where ξ_i represents corrective measure and C is constant defining the size of inaccuracy allowed.

4.2 Decision Trees

Decision tree learning maps observations about the training data to a tree structure that is then used for samples classification. It is one of the predictive modeling approaches used

in statistics, data mining and machine learning. Main advantages of decision trees are that they are simple to understand and interpret. They are also robust to noisiness in the data and performs quite well with large datasets.

There is two type of nodes in decision trees: decision nodes (each internal node) and leaf nodes. When predicting classification of record $s \in S$, decision node compares a value of feature $f_j(s)$ with threshold t and determines next direction of decision making in the tree. Leaf nodes indicate final classification $c \in C$ of record s.

When building a single decision tree we use so called greedy algorithm. A decision node is created by selecting feature $f_j \in F$ and the threshold value t that splits the S to:

$$S_j = \{s \in S | f_j(s) > t\}$$

$$S_j^{'} = \{s \in S | f_j(s) \leq t\}$$

Feature f_j and threshold t are chosen with respect to minimize error value $E_{j,t}$.

$$E_{j,t} = \frac{1}{|S_j|} \sum_{i:s_i \in S_j} I(c_i \neq c) + \frac{1}{|S_j^{'}|} \sum_{i:s_i \in S_j^{'}} I(c_i \neq c).$$

Where error for each subset is computed as the number of misclassified records, in case that we would classify all records in subset by class that has majority there, divided by size of the subset. Each of newly created subsets $S_j, S_j^{'}$ is then split in the same way, and all approach is repeated until each subset does not contain records of only one class c.

4.3 Random Forests

Random forest, presented by Breiman in 2001 [6], works as a large collection of decorrelated decision trees. It also uses averaging to improve the predictive accuracy and control over-fitting. Each tree is created from different random sub-sample of the dataset. When classifying new sample each tree predicts the class of the sample. The class which has the majority of all tree decisions in the forest is the final decision for a sample.

Follows an example of how we grow a simple random forest with M trees. From our learning dataset:

$$L = \begin{bmatrix} \mathbf{v}_1 & y_1 \\ \mathbf{v}_2 & y_2 \\ \vdots & \vdots \\ \mathbf{v}_N & y_N \end{bmatrix}$$

we create M subsets $L_1, L_2, \cdots, L_M$ of the same size $|L_1| = |L_2| = \cdots = |L_M|$, by choosing samples randomly with replacement:

$$L_1 = \begin{bmatrix} \mathbf{v}_1 & y_1 \\ \mathbf{v}_2 & y_2 \\ \vdots & \vdots \\ \mathbf{v}_{139} & y_{139} \end{bmatrix}, L_2 = \begin{bmatrix} \mathbf{v}_2 & y_2 \\ \mathbf{v}_{50} & y_{50} \\ \vdots & \vdots \\ \mathbf{v}_{200} & y_{200} \end{bmatrix}, \ldots, L_M = \begin{bmatrix} \mathbf{v}_{50} & y_{50} \\ \mathbf{v}_{61} & y_{61} \\ \vdots & \vdots \\ \mathbf{v}_{155} & y_{155} \end{bmatrix}$$

From these datasets the single decision trees are then created and each of them gives its result for final classification of sample. Figure 4.2 shows example of three simple decision trees.

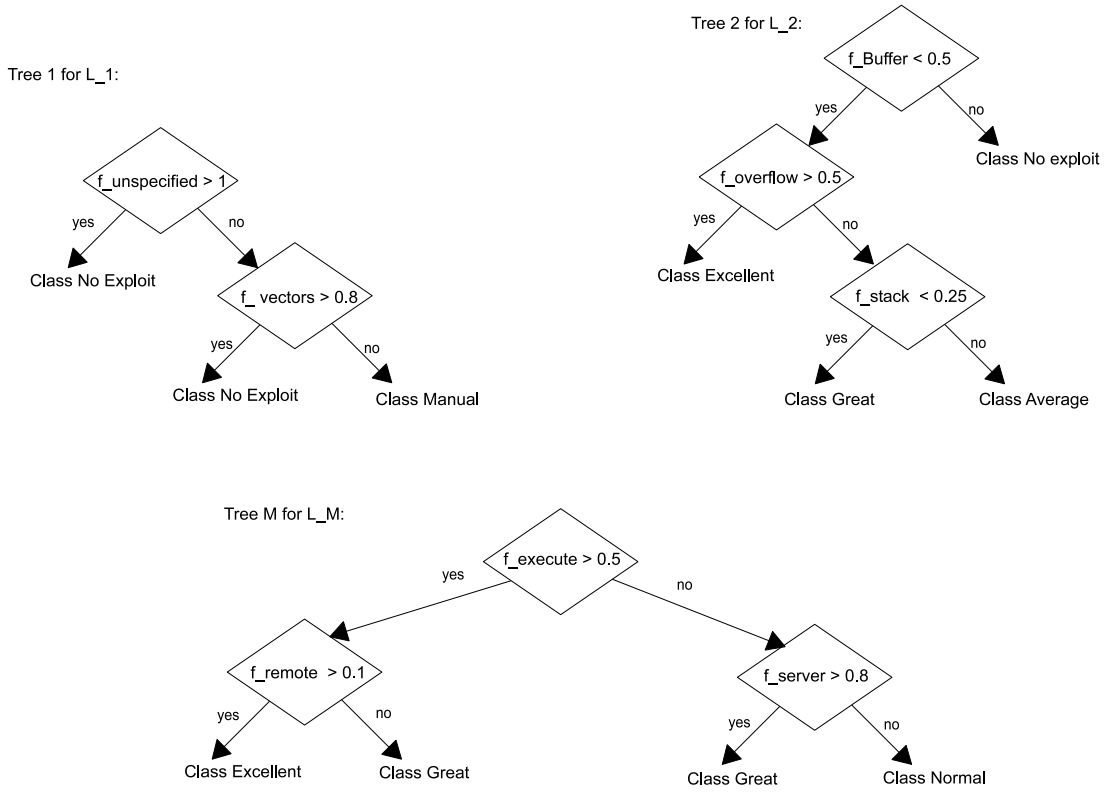


Figure 4.2: Decision trees

In our project we did not want to stick only with the SVM classifiers, therefore, we have also used Random forest classifier for predicting. Thanks to a larger size of our data sample we have been able to create the forests with 100 trees.

Chapter 5

Experiments and Results

5.1 Implementation

For implementing scripts and algorithms we chose Python, mostly because it is a simple and comprehensive, yet powerful tool. It has a great system of scientific libraries available, and interacts well with other platforms (e.g. SQLite).

5.1.1 SQLite

Database of vectors was created with the use of the SQlite library, which allowed storing data in a simple and easily accessible SQL-engine-based way. Our python script for database handling is attached Appendix A.

5.1.2 Scikit Learn

For implementing machine learning algorithms we used the Scikit-learn library [15]. Scikit-learn offers a large amount of state-of-the-art implementations of many machine learning algorithms, and tools for their evaluations which were suitable for our project. The description of used methods and models follows.

- `svm.LinearSVC` - support vector machine classifier implemented on terms of `LIBLINEAR` an open source library for large-scale linear classification [10]. We used classifier with parameter `class_weight` set to `auto`, thus the classifier assigns weights for classes according to amount of their instances in dataset.
- `RandomForestClassifier` - estimator that fits a number of decision tree classifiers on various subsamples of the dataset. We used it with 100 trees by setting the parameter `n_estimators` and the parameter `class_weight` was set to `auto` the same way as in the previous example.
- `cross_validation` - method `cross_val_predict(classifier,vectors,labels)` was used for training and testing estimators performance, on terms of cross validation.

- `TfidfTransformer - fit_transform(vectors)` allowed us to normalize vectors according to *Term-frequency–Inverse-document-frequency* normalization method.
- `CountVectorizer` - class that transforms data vulnerabilities descriptions represented in 2-dimensional data arrays into vectored bag of words representation by function `fit_transform()`.
- `metrics` - class that provides us methods like `accuracy_score(), roc_auc_score(), roc_curve()` for classifiers quality measures evaluation.

5.1.3 Natural Language Toolkit

Natural Language Toolkit is a platform for working with human language data in python. It provides functions as tagging, stemming, semantic reasoning and parsing. We used it in vector creation when extracting data from NVD vulnerability feeds and also when resolving negations in a text.

- `stopwords` - array of common stop words such as *the, a, and* which we separated from vulnerability descriptions.
- `pos_tag(words)` - processes a sequence of words, and assigns a part-of-speech tag to each word, for us those meaning verb (with tag `VB`) were important (taken from chapter 5 in [4]).

5.2 Reciever Operating Characteristics

Receiver Operating Characteristics is a tool for evaluating a performance of a binary classifier. It is typically displayed in a form of graphical plot. Although ROC characteristic is used often with continuous classifiers, we use it in our case to analyze the relations between true positive rate (TPR) and false positive rate (FPR) in a structured form. The ROC curve is created by plotting the FPR on *x-axis* and the TPR (also known as sensitivity) on *y-axis*. Where:

$$\mathrm{TPR} = \frac{\sum \text{True positive}}{\sum \text{Predicted positive}}, \mathrm{FPR} = \frac{\sum \text{False positive}}{\sum \text{Predicted negative}}.$$

In the problem, where we have to deal with multiple classes we compute ROC score for each class separately, by making the results binary (desired class is considered as positive and the rest of classes as negative). That allows us to characterize the sensitivity of classifier for each class.

Area under the ROC curve (referred as AUC score) is way to display ROC characteristic in numbers. In general we consider binary classifier as:

AUC	**evaluation**
$\langle 0.9, 1)$	excellent
$\langle 0.8, 0.9)$	great
$\langle 0.7, 0.8)$	very good
$\langle 0.6, 0.7)$	good
$\langle 0.5, 0.6)$	sufficient
< 0.5	insufficient

5.3 Baseline Approach

For a better analysis of our classifiers results, we created two baseline approaches and computed their accuracy and ROC score. The first one predicts the exploit rank randomly, the second one proportionally with respect to a number of samples for particular classes in the training dataset. Average accuracy of random approach was **13.32%** and ROC score was **0.49** and accuracy of proportional approach was **22.56%** and roc score **0.50**.

5.4 Experiments

For estimating rank, we chose Linear SVM and Random Forest Classifier. Both of them were tested with each of the mentioned normalization methods. Whole training and testing process was performed via cross validation, with five folds, and was also run five times to avoid inaccurate results.

5.4.1 Linear SVM

Results show accuracy of tested SVMs, ROC score. All tested SVMs were tuned with respect to constant C (C is the size of inaccuracy allowed when classifying noisy samples, mentioned in 4.1.1).

Normalization	C	Accuracy	ROC score	Time(s)
TF-IDF	0.02	34.35%	0.59	0.81
sum	0.02	32.55%	0.52	2.83
max	0.06	30.44%	0.55	4.74

Table 5.1: Results of SVM classifiers.

The best classification was performed by SVM with TF-IDF normalization method. It predicted rank with over 20% higher accuracy against random assignment and 10% against proportional assignment of classes. This was expected because TF-IDF normalization method is widely recommended when working with text representation. ROC curves for classifiers are shown in Figure 5.5.2.

Important is to notice that even though some classifiers predict with higher accuracy, their ROC-AUC weighted score is lower. It means they probably generalize the prediction, and favor those ranks with greater frequency in the dataset.

5.4.2 Random Forest

Random forest classifiers were trained with one hundred number of trees. There was almost no difference over normalization methods in accuracy, ROC score or time complexity. Generally, Random Forests classifiers over performed SVMs in accuracy with the difference of 8-12%. Overall results are shown in Table 5.2, ROC curves for each class in Figure 5.5.2.

Normalization	Accuracy	ROC score	Time(s)
TF-IDF	43.57%	0.58	19.32
sum	42.51%	0.58	19.26
max	42.61%	0.58	20.76

Table 5.2: Results of Random forest classifiers.

5.5 Experiments with Language Processing

Here we present results of experiments on the dataset created with replacing the negated verbs for new features. We used same classification algorithms as in the previous example. The whole training and testing process was also performed with use of cross-validation, with five folds, and run five times.

5.5.1 Linear SVM

Normalization	C	Accuracy	ROC score	Time(s)
tf-idf	0.09	34.89%	0.59	0.73
sum	0.02	32.70%	0.52	2.77
max	0.16	29.73%	0.55	4.51

Table 5.3: Results of SVM classifiers with language processing.

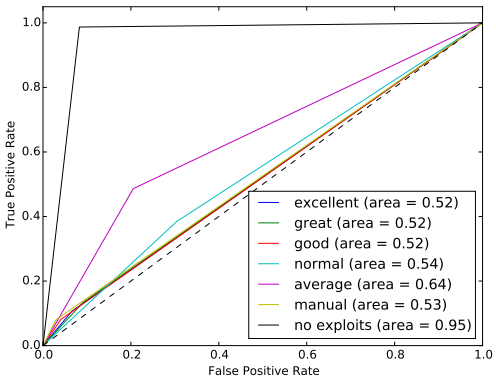
Classifier with normalization method was the best one as in previous example. The overall accuracy of classifiers remained the same, no significant improvements were observed. ROC curves for classifiers are shown in Figure 5.5.2.

5.5.2 Random Forest

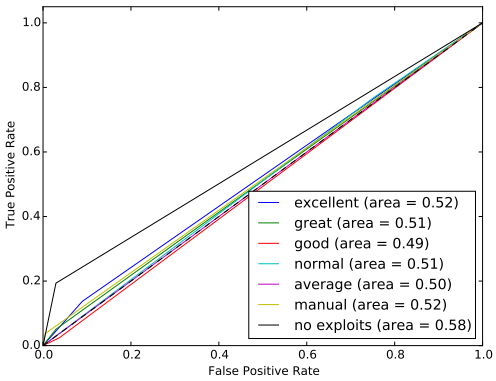
Results of random forest classifiers with use of language processing. ROC curves for classifiers are shown in Figure 5.5.2.

Normalization	Accuracy	ROC score	Time(s)
tf-idf	42.81%	0.58	19.06
sum	42.88%	0.58	18.89
max	43.44%	0.58	18.72

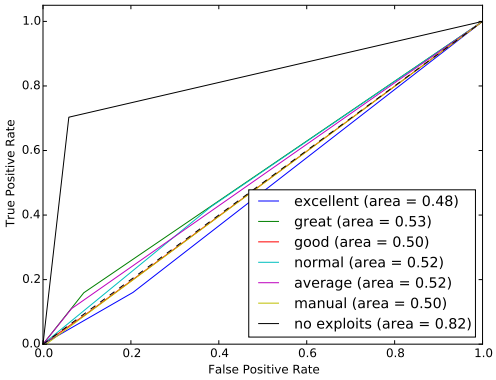
Table 5.4: Results of Random forest classifiers with use of language processing.



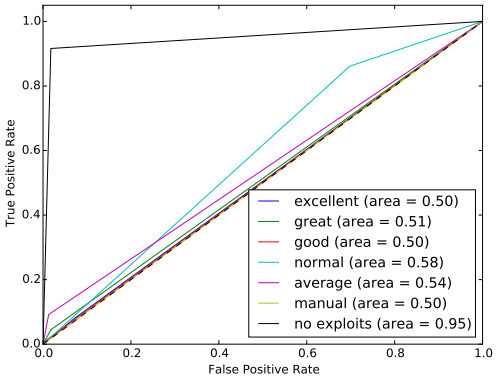
SVM norm: TF-IDF



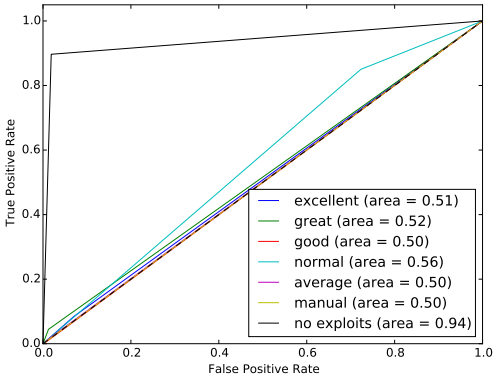
SVM norm: sum



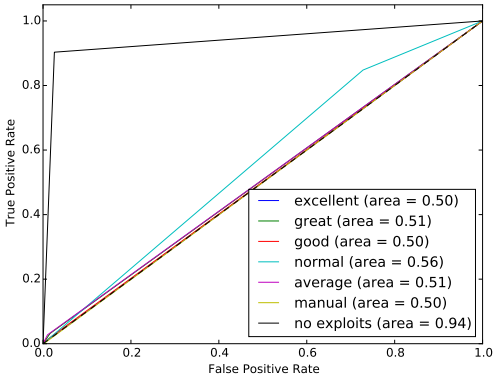
SVM norm: max



RandomForest norm: TF-IDF

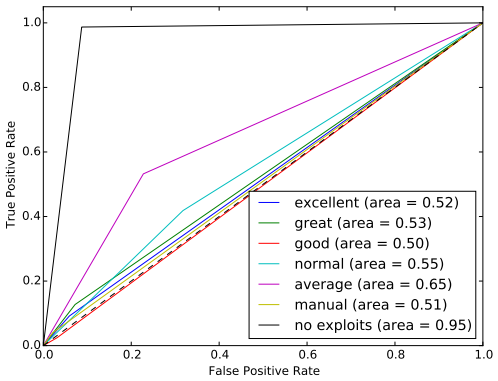


RandomForest norm: sum

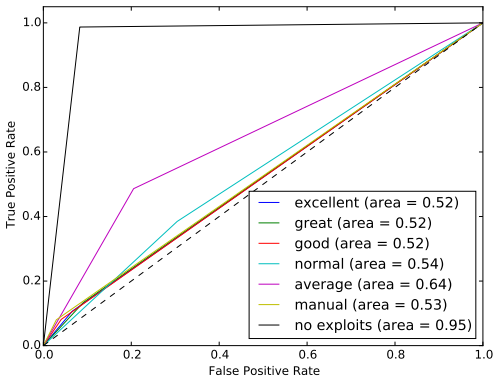


RandomForest norm: max

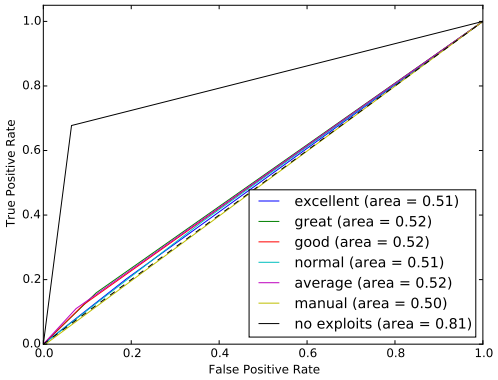
Figure 5.5: ROC curves (no language processing).



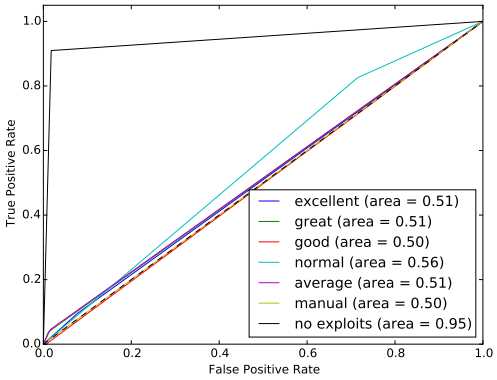
SVM norm: TF-IDF



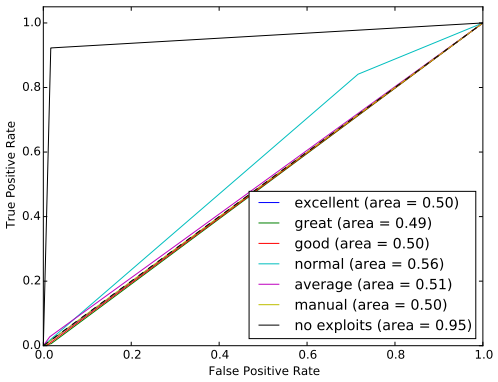
SVM norm: sum



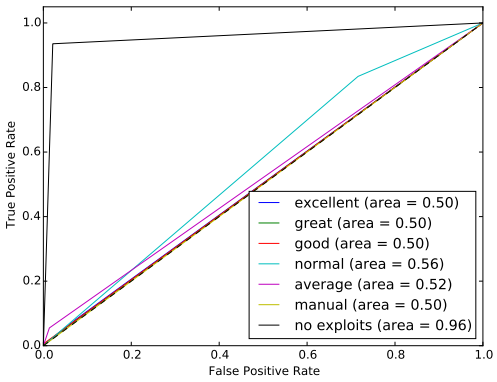
SVM norm: max



RandomForest norm: TF-IDF



RandomForest norm: sum



RandomForest norm: max

Figure 5.6: ROC curves (with language processing).

5.6 Exploit Existence Prediction

In addition to our experiments, we try to predict only the existence of exploit. As we mentioned in 2.3, we found 1123 vulnerability records that have an exploit in Metasploit. We use all of them as positive samples and added the same number, 1123 randomly chosen vulnerabilities with no existing exploit, as negative samples. We test Random Forest and SVM classifiers five times with five-fold cross-validation and evaluate their average accuracy. Baseline approaches for comparison have both (proportional and random assigning of labels) the average accuracy of 50%.

Normalization	Accuracy
TF-IDF	93.09%
sum	64.78%
max	94.43%

Table 5.7: Results of Linear SVM classifiers with different normalization methods.

Normalization	Accuracy
TF-IDF	97.19%
sum	97.39%
max	97.22%

Table 5.8: Results of Random Forest classifiers with different normalization methods.

Chapter 6

Result Discussion

6.1 Achieved Accuracy

The best SVM classifier predicted rank with accuracy **34.89** %, Random Forest classifiers were generally better, in the margin about 10%, the best one predicted ranks with **43.57**% accuracy. In comparison to base approaches, we predicted rank with over 30% higher accuracy against random assignment and 20% against proportional assignment of labels.

6.2 Sensitivity, ROC curves analysis

Each classifier was sensitive to *no exploit* class and worked well when predicting it. Regarding other classes, SVM with TF-IDF normalization and Random forest classifiers (with each of norm. methods) showed at least some sensitivity. SVM was good also in predicting class *average* and Random forests were more sensitive when predicting class *normal*. Sensitivity for other classes (*manual, good, great, excellent*) was only slightly better than in baseline approaches.

6.3 Language Processing Contribution

From our results, it is obvious that using of language processing has no significant effect on successful prediction. This might be because with such large amount of features, changing one of them has not a big influence on the overall classification. Another reason might be that the amount of negated sentences was 368, we were able to determine the correct verb (connected with negation) only for 222 of them. It was caused mostly by the unnatural composition of sentences in vulnerability summaries, where we found a lot of abbreviations and references to different software, which is hard to be found in natural language dictionaries for processing.

6.4 Predicting Exploit Existence

Significantly greater accuracy was achieved when predicting only the existence of exploit, **97.22**%. This success comes from high sensitivity of our classifiers for *no exploit* class. From tested classifiers Random Forest were slightly better, with no regard to normalization method. SVM with TF-IDF normalization performed the best among SVMs, followed by *max* normalization, *sum* normalization performed the worst.

This test also confirmed results previous results of previous work in [5],[8], that we are able to predict exploit existence accurately with this kind of approaches and proved that use of ML in vulnerability prioritization can be very useful.

6.5 Feature Weights Inspection

In this section we examined weights of features coming from bag-of-word representation, to determine features that have the biggest influence on predicting that an individual sample belongs to a certain class. Weights were inspected for LinearSVM (with normalization by TF-IDF) which had the best ROC score and accuracy among tested SVMs with no language preprocessing. Before examining the most important features, all weights were normalized by their occurrence in the dataset according to $\overline{w}_j = w_j(N_j/N)$. For each label top ten negative and positive influential features are shown in the Figure 6.1.

From results we tried to derive a few insights:

- The vulnerability which allows code execution is more likely to have an existing exploit because word *execute* occurred two times as the indicator that vulnerability will have an exploit and one time as an indicator of non-existing exploit (but with small weight). These vulnerabilities might be more attractive for adversaries or also they were probably already recorded with existing exploit.
- Remote access to vulnerability also means bigger danger. It is obvious that attackers will look for those vulnerabilities in the targeted system, instead of those that require personal access to the targeted system. The occurrence of word *remote* proves this, when four times showing as exploit indicator.
- When vulnerability enables an adversary to create a buffer overflow attack, it should most likely be treated as important. At least that what we can judge by both words *buffer* and *overflow* appearing as top positive indicators for class *excellent*.
- When the way, how to exploit the vulnerability, is uncertain, there probably will be no exploit ever created. We derive this from the occurrence of *unspecified* only as a positive indicator for *no exploit* label and negative for three other classes. This idea is proved by the occurrence of word *vector* (which is present as an indicator of non-existing exploit two times).

Although some other features occur frequently in the top influential list among classes, they cannot be strictly referred as indicating an existence of exploit or any other characteristic of a vulnerability. Hence, we are not able to deduce any conclusion about their meaning.

CLASS excellent

name:	norm. w:	name:	norm.w:
execute	0.006920	vulnerability	-0.004951
attackers	0.005790	windows	-0.004027
code	0.005022	users	-0.002518
overflow	0.004926	file_	-0.002465
cve	0.004848	stack	-0.001694
earlier	0.002935	possibly	-0.001414
allows	0.002535	application	-0.001265
arbitrary	0.002523	sp	-0.001260
server	0.002071	service	-0.001181
buffer	0.001631	string	-0.001153

CLASS great

name:	norm. w:	name:	norm.w:
sp	0.004279	vulnerability	-0.003463
windows	0.003658	cve	-0.002775
remote	0.003644	allows	-0.002300
service	0.003315	vectors	-0.001848
code	0.002362	unspecified	-0.001526
based	0.002255	php	-0.001505
stack	0.002045	users	-0.001499
aka	0.001732	arbitrary	-0.001474
denial	0.001646	overflow	-0.001281
cause	0.001623	execute	-0.001091

CLASS good

name:	norm. w:	name:	norm.w:
attackers	0.002675	arbitrary	-0.006720
windows	0.001587	allows	-0.006412
authentication	0.001425	execute	-0.005411
demonstrated	0.001118	code	-0.003578
properly	0.001077	unspecified	-0.002958
allow	0.001038	vulnerability	-0.002654
api	0.000916	users	-0.002612
microsoft	0.000822	earlier	-0.002579
requests	0.000761	crafted	-0.002564
access	0.000746	buffer	-0.002510

CLASS normal

name:	norm. w:	name:	norm.w:
code	0.004789	sp	-0.003333
overflow	0.004070	earlier	-0.003003
buffer	0.003208	server	-0.002880
service	0.003039	aka	-0.002108
long	0.002820	control	-0.001455
stack	0.002401	windows	-0.001445
note	0.002263	multiple	-0.001365
request	0.001947	unspecified	-0.001190
cause	0.001753	allow	-0.001069
allows	0.001644	activex	-0.001060

CLASS average

name:	norm. w:	name:	norm.w:
service	0.009559	code	-0.008767
crafted	0.009489	windows	-0.008239
users	0.007562	server	-0.008080
remote	0.006130	aka	-0.007588
php	0.005757	earlier	-0.006320
rash	0.005730	sp	-0.005621
string	0.005119	attackers	-0.005611
adobe	0.003331	buffer	-0.004897
files	0.002625	based	-0.004801
multiple	0.002373	vulnerability	-0.004747

CLASS manual

name:	norm. w:	name:	norm.w:
php	0.005862	attackers	-0.004675
function	0.004787	service	-0.004229
windows	0.004262	overflow	-0.002621
file	0.003356	cause	-0.002555
sp	0.002845	denial	-0.002548
server	0.002299	buffer	-0.002386
users	0.001974	code	-0.001970
remote	0.001414	earlier	-0.001398
authenticated	0.001287	based	-0.001361
commands	0.001162	method	-0.001327

CLASS no exploits

name:	norm. w:	name:	norm.w:
vulnerability	0.006543	code	-0.004637
allows	0.005162	windows	-0.003616
unspecified	0.004072	execute	-0.002848
vectors	0.002725	overflow	-0.002460
earlier	0.001956	buffer	-0.002450
aka	0.001950	remote	-0.002122
users	0.001936	long	-0.002073
management	0.001298	function	-0.002019
application	0.001064	sp	-0.001875
method	0.000999	php	-0.001832

Figure 6.1: Most important features for each class

Chapter 7

Conclusion

The aim of our project was to use ML algorithms for predicting exploit ranking. We presented a solution that creates features for vulnerabilities classification by splitting their description by bag-of-words algorithm and then predicts rank by SVM and Random Forest algorithms. The overall accuracy of classifiers ranged between 31-43% and Random forest outperformed SVM in the difference of 10%. When evaluating the final performance, we have to take into consideration the amount of the predicted classes and results of random classifiers. Thus, the final results of rank prediction, even when far from perfect classification, can be considered as successful.

Last part of our tests showed that algorithm can be used when predicting the only existence of exploit, with an accuracy of 97.22%. Great performance of predicting the existence of exploit confirms results of previous works and shows how machine learning can be useful in predicting rank and prioritizing vulnerabilities.

An important observation was that the use of Language processing in vulnerability description parsing, which we expect to improve the solution, did not significantly increased the final accuracy.

There are still many ways how to improve the accuracy of our classifiers. One of them is to gather a greater amount of information for particular vulnerabilities and also larger the amount of training and testing sets. Both would be possible when considering different data sources. Further work also contains implementing automatized software that would require only a vulnerability CVE-id and found all possible information and provide the complete risk assessment.

Bibliography

[1] Common Vulnerabilities and Exposures. <https://cve.mitre.org/>. Accessed: 2016-05-15.

[2] National Vulnerability Database. <https://nvd.nist.gov/>. Accessed: 2016-05-08.

[3] ALLODI, L. – MASSACCI, F. A preliminary analysis of vulnerability scores for attacks in wild: the ekits and sym datasets. In *Proceedings of the 2012 ACM Workshop on Building analysis datasets and gathering experience returns for security*, p. 17–24. ACM, 2012.

[4] BIRD, S. – KLEIN, E. – LOPER, E. *Natural language processing with Python.* " O'Reilly Media, Inc.", 2009.

[5] BOZORGI, M. et al. Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining.* 2010, p. 105–114.

[6] BREIMAN, L. Random forests. *Machine learning.* 2001, 45, 1, p. 5–32.

[7] BURGES, C. J. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery.* 1998, 2, 2, p. 121–167.

[8] EDKRANTZ, M. – SAID, A. Predicting Cyber Vulnerability Exploits with Machine Learning. In *Thirteenth Scandinavian Conference on Artificial Intelligence: SCAI 2015*, 278, p. 48. IOS Press, 2015.

[9] ESCHELBECK, G. The Laws of Vulnerabilities: Which security vulnerabilities really matter? *Information Security Technical Report.* 2005, 10, 4, p. 213–219.

[10] FAN, R.-E. et al. LIBLINEAR: A library for large linear classification. *The Journal of Machine Learning Research.* 2008, 9, p. 1871–1874.

[11] FREI, S. et al. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, p. 131–138. ACM, 2006.

[12] FRUHWIRTH, C. – MANNISTO, T. Improving CVSS-based vulnerability prioritization and response with context information. In *Proceedings of the 2009 3rd international Symposium on Empirical Software Engineering and Measurement*, p. 535–544. IEEE Computer Society, 2009.

[13] HARTANTO, T. Penetration Testing: Testing the Security of Computer Systems. *CS2107-Semester IV*. 2014, p. 65.

[14] LABUť, M. Estimating the Attacker's Cost for Exploiting Computer Network Vulnerabilities. Master's thesis, Czech Technical University, 2016.

[15] PEDREGOSA, F. et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, 12, p. 2825–2830.

[16] RIEKE, R. Modelling and analysing network security policies in a given vulnerability setting. In *Critical Information Infrastructures Security*. Springer, 2006. p. 67–78.

Appendix A

Content of the CD

- `/Code/` – source code for creating datasets and testing classifiers
- `/Datasets/` – datasets for learning and testing performance of classifiers
- `/gavenkar_BP_2016.pdf` – the electronic version of this work