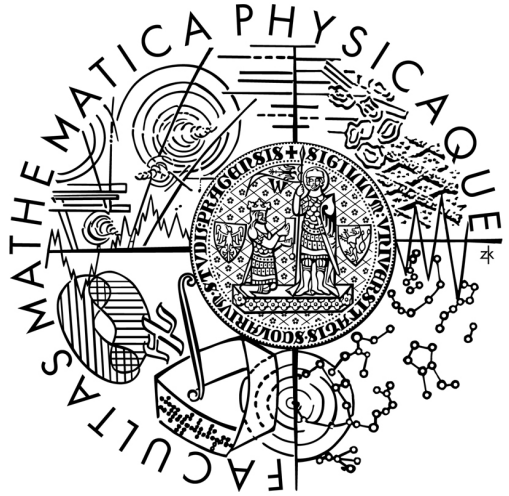
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Dušan Domány

Vulnerability Reports Analysis and Management

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND MATHEMATICAL LOGIC

Supervisor of the master thesis: Mgr. Daniel Toropila

Study program: Computer Science

Specialization: Software Systems

Prague 2011

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisor Mgr. Daniel Toropila, who guided me throughout the whole process of this thesis work. I deeply appreciate his comments and ideas that helped improving the quality of this thesis a lot.

Many thanks to Privatdoz. Dipl.-Ing. Edgar Weippl as well, who has inspired the idea and provided me with invaluable help during the initial steps of this work.

I would also like to thank all the people that participated on this work by providing advice and study materials. My special thanks goes to Secure Business Austria where many of these people work.

Last but not least, I would like to thank my family and my girlfriend as their encouragement and support were essential for this thesis to come to a successful end.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 4, 2011 Dušan Domány

Abstract

Název práce: Analýza a správa hlášení o bezpečnostních slabinách

Autor: Dušan Domány

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Daniel Toropila, KTIML MFF UK

Abstrakt: Různé slabiny v softvérových produktech můžou často představovat značnou bezpečnostní hrozbu jestliže jsou objeveny nebezpečnými útočníky. Je proto důležité tyto slabiny identifikovat a ohlásit jejich existenci zodpovědným osobám dřív než jsou zneužity nebezpečnými subjekty. V průběhu posledního desetiletí počet bezpečnostních hlášení o objevených slabinách v různých softvérových produktech rapidně vzrostl. Stává se stále naročnejším zpracovávat všechny tyto hlášení manuálně. Tato práce rozebírá různé metody, které je možné použít pro automatizaci několika důležitých procesů při sbíraní reportů a jejich třídení. Reporty jsou analyzované různými způsoby, které zahrnují techniky text miningu, a výsledky této analýzy jsou aplikovány ve formě praktické implementace.

Klíčová slova: informační bezpečnost, softvérová slabina, bezpečnostní hlášení, text mining, strojové učení

Title: Vulnerability Reports Analysis and Management

Author: Dušan Domány

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Daniel Toropila, KTIML MFF UK

Abstract: Various vulnerabilities in software products can often represent a significant security threat if they are discovered by malicious attackers. It is therefore important to identify these vulnerabilities and report their presence to responsible persons before they are exploited by malicious subjects. The number of security reports about discovered vulnerabilities in various software products has grown rapidly over the last decade. It is becoming more and more difficult to process all of the incoming reports manually. This work discusses various methods that can be used to automate several important processes in collecting and sorting the reports. The reports are analyzed in various ways, including techniques of text mining, and the results of the analysis are applied in form of practical implementation.

Keywords: information security, software vulnerability, security report, security advisory, text mining, machine learning

Contents

1 Introduction

The importance of information technologies has grown rapidly over the last few decades. Computers have become a common property of many families and an essential part of the industry. In many countries, a connection to the Internet is considered to be a matter of course.

Human society is becoming more and more dependent on the correct and uninterrupted operation of increasingly complex systems. The complexity of modern information and communication systems and the related difficulty to ensure an adequate control over their operation creates many opportunities for targeted attacks.

In order to prevent these attacks from happening, it is necessary to identify all potential vulnerabilities and implement adequate measures before they are found and exploited by malicious attackers. Many of these vulnerabilities can occur in various software products. If a vulnerable software product is widely used, its vulnerability can represent a serious security threat for many people and organizations. It is therefore important that these vulnerabilities are identified as soon as possible and adequately reported to responsible persons.

Because of the increasing number of security reports, it is often not feasible to manually process all of the incoming reports and correctly evaluate their relevance to a particular information system.

The following work discusses and implements various methods that can be used to automatically collect security reports, analyze them, and allow efficient extraction of relevant information. The analysis part focuses on applying various text mining techniques with the goal of identifying interesting relationships among the collected documents, which, as we show, could help the subsequent filtering of the information.

After the brief introduction in the first chapter, the second chapter starts by describing the field of information security in general and introducing its main aspects. Next, the chapter focuses on the target domain of this thesis which is software security. It explains how vulnerabilities in various software products are discovered and adequately reported to help prevent the damage that could be caused by their targeted exploitation. The last part of this chapter describes several security standards that have been established by the National Institute of Standards and Technology (NIST) in order to standardize the format of the security reports.

Chapter three provides an introduction to the methods of text mining, some of which were used in this thesis in order to perform various analyzes of the security reports. The chapter describes the most common text mining techniques that are used to transform an unstructured text into a form suitable for an automated analysis. Then it continues with the introduction of various text mining tasks, as well as several algorithms that are commonly used to fulfill those tasks.

Chapter four is the central chapter of this thesis as it describes the analysis that was carried out in order to identify interesting relationships among the existing security reports. It starts by introducing Apache Lucene and the GATE text mining framework, both of which were the primary tools used in this work. The chapter then continues with describing the first part of the analysis which was mostly done by human observation and study of the related security standards.

The second part of the analysis described in the fourth chapter focuses on one

of the most common text mining tasks, which is the text classification. The goal of this part of the analysis was to classify software security reports based on the common software weaknesses listed in the Common Weakness Enumeration standard. The text describes how various text mining methods and algorithms were evaluated and compared to each other in order to achieve the best possible results.

The last part of the analysis focuses on another common text mining task, which is the information extraction. The text describes various methods that were applied in order to extract the key information from security reports, including dictionary search, knowledge-based rules, and machine learning.

Chapter five describes the major steps taken in order to collect documents for the purposes of this thesis. The text focuses on describing some non-trivial tasks that were carried out to gather the documents and prepare them for automated processing. The second part of the chapter introduces the Advisory Updaters tool which was implemented as a part of this work to simplify the process of gathering and preprocessing documents from various sources. The tool already includes implementation of the mechanisms for collecting documents from six sources that are also described in this chapter.

Chapter six describes the Vulnerability Management Tool application which was also implemented as an integral part of this work. The application allows its user to display, analyze, and filter the documents collected by the Advisory Updaters tool using a graphical user interface. It integrates the results of the analysis described in chapter four and provides an implementation that can be used by an average user without any deeper knowledge of the underlying methods. The text includes a user guide together with an introduction to the technical background of the application.

The last chapter contains the final conclusions and several ideas for the future work.

2 Information Security

2.1 Overview

Security has always played an important role in people's lives. Feeling safe and being safe is one of the fundamental needs of human beings. If a society wants to survive and prosper, it needs to defend its members from the dangers that come from the outside and the dangers that come from the inside. Whether it comes to individuals, families, companies, or countries, the lack of security almost always leads to undesired and possibly catastrophic events. Although the most critical requirement is to ensure that the human lives are not threatened, it is also very important to provide sufficient protection of property and people's interests.

These ideas have been integrated not only to the society as a whole, but also to the various sectors of industry. The necessity for security and reliability has been recognized by most engineering disciplines.

In addition to lives and material property, it is also often important to protect intangible assets, like information. Being in a possession of a particular piece of information, and being able to protect it properly, can often provide an organization the necessary advantage needed to improve its competitiveness.

Nowadays, information can have a real value and can be traded just like any other property. Its importance has grown even more with the advent of computer technology. It is the role of information security to ensure protection of information and its undisturbed transmission to authorized persons.

Basic Terms

We will first introduce some basic terms to ensure proper understanding of the text that follows.

An **information system** is a collection of hardware, software, data, people and procedures used to gather, transfer, store, and process information. Any part that is relevant from the perspective of information security is considered to be an **asset**. Any potential act or event that could interact with the assets in an undesired way is considered to be a **security threat**. Components that are designed to protect the assets against the security threats are referred to as **security measures**.

An **object** is a piece of data, or other part of the information system, which can be read, manipulated, used, or altered. A **subject** is an entity (usually a person or a process) which can access these objects in some of the listed ways. A **malicious subject** is an entity which performs its activities within the information system in a way that deviates from the established rules.

A computer **virus** is a malicious piece of software that can attach itself to other program and secretly executes when the host program is run. It typically spreads among multiple programs by making exact copies of itself every time it executes. Once a virus is executing, it can perform whatever malicious activity it was designed for.

A **worm** is a program that can propagate itself over a computer network by sending its own copies to the other machines in the network. Once a copy arrives and is executed, it performs its designed activity on the target machine and continues in the propagation.

Besides various computer programs, malicious subjects can be also people. These people typically try to take advantage of the weak spots of an information system to get possession of some secret information, inflict damage, or use the information system to their advantage. These people are typically referred to as **hackers**.

To protect itself against these various malicious subjects and other dangers, an organization needs to establish a document called **security policy**. This document defines the competencies, responsibilities, rules, and principles which the organization and its staff should follow to achieve the necessary security objectives.

Some of the typical security objectives are described in the following sections.

Operation Reliability

The operation reliability of an information system often determines the ability of its users to carry out their tasks, meet their obligations to other subjects, make the right decisions, etc. The suspension of its operation, or other deviation from the projected or anticipated activity (e.g. incorrect data processing), has undoubtedly a negative effect on their work. The importance of the operation reliability often rapidly grows during emergency situations, which can be caused for example by an unexpected event, or a failure of a critical component.

Ensuring a reliable operation of the information system means an implementation of measures, which allow the information system to provide its services to the users on time and of the required quality, even in very unusual situations.

Data Protection

Every piece of data can be associated with some descriptive attributes, like accuracy, correctness, completeness, reliability, relevance, etc. Violation of any of these attributes is called a security incident.

Ensuring data protection means an implementation of measures, which prevent an unauthorized modification of attributes that are considered to be important by the owner of the data, or by another authorized subject.

Although the choice of the important set can be highly individual, it typically includes the following attributes:

- **Confidentiality:** Assurance that private or confidential information is not made available or disclosed to unauthorized individuals.
- **Integrity:** Assurance that information and programs are changed only in a specified and authorized manner.
- **Availability:** Assurance that systems work promptly and service is not denied to authorized users.

It is very important when formulating the requirements for data protection to correctly determine which data and which of their attributes must be protected. A commonly overlooked need is the protection of meta-data, like user passwords, encryption keys, etc. Unauthorized manipulation of such data may result in significant damage.

Protection of Users' Interests

The protection of users' interests mainly involves protecting their privacy in connection with their activities and interaction with the information system. This includes providing:

- **Anonymity:** A state in which the user can make use of the functionality of the information system without revealing his identity to a particular group of subjects.
- **Pseudonimity:** A state in which the user can make use of the functionality of the information system without revealing his identity to a particular group of subjects, while the user is still accountable for his actions within the information system.
- **Unlinkability:** A state in which the user can make use of multiple resources or services of the information system without risking that someone else would be able to link these activities together.

Securing an Information System

These requirements may seem to be straightforward, but the mechanisms used to meet them can be rather complex and properly securing an information system can be a very challenging task. It requires knowledge, experience and a considerable amount of resources. Developing reliable security measures requires consideration of many potential attacks. Having the security measures designed, it is necessary to decide how to integrate them into the architecture of a particular information system in both physical and logical sense.

"*The great advantage that the attacker has is that he or she need only find a single weakness while the designer must find and eliminate all weaknesses to achieve perfect security.*"[STAL08]

There are security norms, which can assist in the process of creating the security policy, like TCSEC, ITSEC, Common Criteria, and BS7799 [BENE07]. However, each information system is different and has its own risk factors. It needs to be considered which assets need to be protected and how. Information security includes various domains, like physical security, software security, cryptography, personal security, and many others. All of these need to be taken into account to build a solid protection. The next chapter introduces the target domain of this thesis, which is the software security.

2.2 Software Security

Software is an essential part of any computer system. It can be the operating system that a particular computer is running, a chat application, or just a simple editor. It is always something that participates on forming the complex architecture of components, which allow the computer to provide the desired services.

Computer components are usually designed to perform some specific task. Whether it is communication via the Internet or music playing, it is something that has an obvious meaning to the person that makes some use of it. However, the

behavior of a particular component can vary depending on the different circumstances. These circumstances can sometimes deviate from what is considered to be normal or common, and potentially trigger an unexpected behavior. This may result into an undesired situation, especially in case that it is deliberately induced by a malicious subject. If the component is a piece of software and the situation represents a security threat, the software is considered to be vulnerable and its usage can be potentially dangerous.

Many software vulnerabilities occur as a consequence of insufficient checking and validation of data and error codes in programs.

"*When writing a program, programmers typically focus on what is needed to solve whatever problem the program addresses. Hence their attention is on the steps needed for success and the normal flow of execution of the program rather than considering every potential point of failure.*" [STAL08]

Software Errors

An error in a program that can cause an unintended behavior is often referred to as a bug. Bugs are generally unpleasant, because they can interrupt the normal interaction between the user and the application. The stability of the application is usually judged by the number of bugs that can be seen. Programmers usually focus on eliminating as many of these bugs as possible, because they influence directly the users' opinion about the application.

In order to make it possible to fix an error, it is first needed to find and identify it. There are various testing techniques that programmers use to accomplish this task, including:

- **Black box testing:** A form of high level functionality testing without any knowledge of how the application is implemented.
- **White box testing:** A form of testing based on an analysis of the internal implementation and structure of the application.
- **Unit testing:** A form of testing that involves creation of code-based tests for individual classes and methods. The tests can be rerun at any time, which makes it possible to identify bugs introduced by refactoring or other changes in the code.
- many others

In all of these cases programmers make their assumptions about the potential inputs to the application and the environment it executes in. The way they test the application reflects the way they know it is going to be used by a common user. The goal is to minimize the chance of an error during the standard usage. Although this usually ensures the proper functionality, it does not guarantee the security of the application at the same time.

"*Software security differs in that the attacker chooses the probability distribution, targeting specific bugs that result in a failure that can be exploited by the attacker. These bugs may often be triggered by inputs that differ dramatically from what is usually expected and hence are unlikely to be identified by common testing*

approaches. Writing secure, safe code requires attention to all aspects of how a program executes, the environment it executes in, and the type of data it processes." [STAL08]

Defensive Programming

To provide more reliable and secure applications, programmers have developed a concept called *defensive programming*. It is also sometimes referred to as *secure programming*.

Defensive programming requires a changed mindset to traditional programming practices, which focus on solving a particular problem. The defensive programmer needs an awareness of the consequences of failure and the techniques used by attackers. He has to understand how failures can occur and the steps needed to reduce the chance of them occurring in the program.

This approach certainly requires much more time, resources and proper education. However, it can pay off in a form of a much more reliable and secure application.

"*Unless software security is a design goal, addressed from the start of program development, a secure program is unlikely to result.*" [STAL08]

Most Common Software Vulnerabilities

This section introduces some of the most common vulnerabilities that can be found in software.

The Buffer Overflow Vulnerability

Buffer overflow can occur when the size of the input provided to the program exceeds the size of the buffer allocated to hold its content. If the size of the input is not checked, an attempt to store the data beyond the limits of the buffer can cause overwriting a part of the memory that is outside the allocated buffer. This part can hold other program variables or program control flow data such as return addresses and pointers to previous stack frames. The buffer can be located on the stack, in the heap, or in the data section of the process. If appropriate security measures are not implemented, this vulnerability can be further exploited by a skilled attacker to execute arbitrary code with the privileges of the attacked process. Even an unsuccessful attempt can often lead to a crash of the application.

The Injection Vulnerability

This kind of flaw occurs when program input data can accidentally or deliberately influence the execution flow of the program itself. One of the most common mechanisms by which this can occur is when the input data are passed as a parameter to another helper program on the system, whose output is then processed and used by the original program.

To deal with this kind of vulnerability, the defensive programmer needs to compare the input data to a pattern that describes their assumed form and reject any input that fails this test.

One of the best known variants of this vulnerability is called *SQL injection*, which allows the attacker to supply a specifically crafted SQL command to the application input to retrieve information from the underlying database.

The Cross-Site Scripting Vulnerability

This kind of vulnerability concerns a case when the input provided to a program by one user is subsequently displayed to another user. It can be most commonly seen in scripted web applications. The attack involves inclusion of a script into the HTML content of the web page that is displayed by the user's browser and eventually executed. It exploits the assumption that all content from one page is equally trusted and is permitted to interact with other content from that page. This can be used to bypass security checks and gain access privileges to sensitive data.

2.3 CERT

Even with the best effort, it is almost never possible to provide a hundred percent security guarantee. Writing secure code requires time and human resources, which are not always available. The market requires that products are delivered in a timely fashion, while not exceeding the specified budget. It is very common that finding all the bugs and fixing them is simply not worth the cost, because the majority of the hidden bugs have often lesser significance than the further delay of the product release. However, even a single undiscovered significant weak spot creates a potential for an attack that can eventually have a catastrophic impact. Table 2.1 lists several historical computer incidents that were enabled by various software vulnerabilities.

Table 2.1: *Estimated damage caused by selected computer viruses and worms. The table is taken from [HENL05]*

Year	Virus or worm	Damage estimation in dollars
1999	virus Melissa	80 million
2000	virus Love Bug	10 billion
2001	worms Code Red I and II	2.6 billion
2001	virus Nimda	590 million to 2 billion
2002	worm Klez	9 billion
2003	worm Slammer	1 billion

The first major incident of this kind occurred in 1988 when the so-called *Morris Worm* hit the Internet. This led to formation of the first *CERT* team that was covered by the U.S. Government. The letters in CERT stand for *Computer Emergency Response Team*. Many English speaking countries rather use the shortcut *CSIRT - Computer Security Incident Response Team*. As the importance of information security grew, many nations and organizations established their own CERT teams. However, the original CERT has still a major role and is known as

CERT Coordination Center (*CERT/CC*).

Generally speaking, CERT attempts to limit the damage caused by attacks on vulnerable software and services by receiving, reviewing, and further propagating the reports about newly discovered vulnerabilities and threats. The reports are propagated via various channels, such as mailing-lists, RSS feeds, and websites.

This naturally raises a question: where does the information originally come from? The first possible source is a security incident that was enabled by a yet unknown vulnerability. If the incident is caused by a malicious software like a worm, then a fast and efficient response is required to ensure that the situation is stabilized as soon as possible and any additional damage is prevented.

However, a preferred approach is to prevent the incident from happening at first place. Several companies have been established over the last few years that do an active research in the area of software security and try to identify vulnerabilities in various software products before they are discovered and exploited by malicious subjects. Some of these companies will be described in Chapter 5.

The information can also come from individuals that call themselves *ethical hackers*. Ethical hackers have a lot in common with traditional hackers. However, the major difference between these two parties is that ethical hackers do not try to take advantage of the found vulnerabilities, but rather submit their discoveries to responsible institutions like CERT. As the importance of ethical hacking grows, so increases also the number of tools that the ethical hackers have at their disposal. These tools include program analysis tools like Valgrind [VALG11], exploitation frameworks like Metasploit [META11], various fuzzers like Peach [PEAC11] and Hotfuzz [HOTF11], SQL injection tools like SQLMap [SQLM11], and many others.

The policy of CERT/CC is to inform the software vendor first and participate on fixing the error. If the vendor does not fix the error within 45 days after it was notified, CERT/CC informs the society about the presence of the threat by publishing an article that describes the vulnerability. CERT/CC usually releases the article also in case that the vendor manages to fix the error, and includes the information that the users can protect themselves by applying the patch.

Nowadays, CERT/CC is far from being the only organization that publishes articles about software vulnerabilities. There are other communities and companies that publish articles of their own, sometimes also providing some paid services.

2.4 Security Standards

The increasing number of the articles has inspired creation of various security standards. Some of them are described in the following subsections.

Security Content Automation Protocol

The *Security Content Automation Protocol* (*SCAP*) is a suite of specifications that standardize the format by which security information about software vulnerabilities is communicated. It is a multipurpose protocol that supports automated vulnerability checking, technical control, compliance activities, and security measurement. Goals for the development of SCAP include standardizing system security management and improving interoperability of security products. The protocol is developed by the *National Institute of Standards and Technology* (*NIST*).

Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (*CVE*) is a SCAP specification that represents a system of assigning unique identifiers to publicly known software vulnerabilities using a common convention. CVE provides a comprehensive list of known software flaws and a globally unique name to identify each vulnerability. The format of a CVE identifier is *CVE-YYYY-NNNN*, where *YYYY* is the year when the vulnerability was disclosed and *NNNN* is a sequential number. The list of software flaws is available as a part of the *National Vulnerability Database*, which will be discussed later in this text. CVE is used in conjunction with other SCAP specifications, like CPE, CVSS, CWE, and OVAL.

Common Platform Enumeration

The purpose of *Common Platform Enumeration* (*CPE*) is to provide consistent names for operating systems, hardware, and applications, which can be used by various parties to reference the software and hardware in a unified and standard way. CPE also provides a dictionary that contains names for many existing products. The format of a CPE name is *cpe:/{part}:{vendor}:{product}:{version}:{update}:{edition}:{language}*. *{part}* determines the type of the product. For example *o* represents an operating system, *a* represents an application, etc. *{vendor}* is the name of the product vendor, *{product}* is the name of the product itself, and the other fields describe additional details about the product version.

Common Weakness Enumeration

Common Weakness Enumeration (*CWE*) provides a comprehensive dictionary that lists many general software weaknesses that are recognized by the security research community. The dictionary is organized in a form of taxonomy, which organizes the weaknesses into categories. Each category can have multiple super-categories and multiple sub-categories. CWE also provides various versions of the dictionary for development and for research purposes.

Common Attack Pattern Enumeration and Classification

Common Attack Pattern Enumeration and Classification (*CAPEC*) provides a dictionary of attack patterns used to exploit software vulnerabilities. Similar to CWE, the dictionary is also organized in a form of taxonomy, which provides a hierarchical classification of the attack patterns.

The Common Vulnerability Scoring System

The *Common Vulnerability Scoring System* (*CVSS*) is a specification for measuring the relative severity of software vulnerabilities. CVSS metrics are divided into three groups: Base metrics measure the fundamental characteristics of vulnerabilities that do not change over time or in different environments, temporal metrics measure those attributes of vulnerabilities that change over time, and environmental metrics measure those vulnerability characteristics that change among various user environments.

The purpose of performing CVSS scoring is to help organizations identify

those vulnerabilities that have the greatest operational impact for them. A detailed description of the metrics and how they are calculated can be found in [CVSS11].

The Open Vulnerability and Assessment Language

The *Open Vulnerability and Assessment Language* (*OVAL*) provides means to encode system details into machine-readable rules that can be used to assess the security state of the system. There are four types of OVAL Definitions: Vulnerability definitions, Patch definitions, Inventory definitions, and Compliance definitions. The standard comes with an implementation, which allows a user to describe the details of his system using the rules mentioned above and have his system evaluated based on the definitions of known vulnerabilities. More information on the standard can be found in [OVAL11].

2.5 Existing Projects

The articles about software vulnerabilities certainly provide interesting material for further research. Proof of that is the existence of several projects which try to analyze the articles and come with new ideas that could help their processing. Information about one of these projects can be found in [WANG09].

One thing to notice about these projects is that they usually use only one or two sources of information. The most commonly used source is the National Vulnerability Database, because it provides the information in a standardized and structured form.

Another observation is that their analysis is usually based only on the structured meta-data that is included in the articles. However, in spite of being completely unstructured, the description text of the vulnerability can be used for automatic processing as well, and can also provide a lot of useful information. There is a relatively new research discipline that focuses on analyzing unstructured text using natural language processing techniques. This discipline, commonly referred to as *text mining*, has recently grown in popularity, as the amount of unstructured data presented on the Internet rapidly increases. The next chapter describes basic concepts of text mining together with some of its important techniques.

3 Text Mining

"*Text mining is a new and exciting area of computer science research that tries to solve the crisis of information overload by combining techniques from data mining, machine learning, natural language processing, information retrieval, and knowledge management.*" [FESA07]

Text mining is a discipline with a focus on deriving useful information from completely unstructured or partially structured texts written in a natural language. It is in many ways similar to data mining. However, the major difference between these two disciplines is that data mining assumes that data have already been stored in a structured format. This assumption is true only if the structure of the data is properly enforced. But even relational databases with their structural design tend to store a lot of information in a form of unstructured text. The text is usually written by people in a way that is generally understandable to other people that speak the same language. The goal of text mining is to simulate this understanding by a machine and use its processing power to analyze the documents and gather interesting information. This raises a question: what is interesting information? The information needs to be considered interesting by people. This makes the question even harder to answer, because people are typically subjective. The answer may usually depend on the target domain of the documents, and potentially on other factors.

Another thing that text mining needs to deal with is that natural language is complex and ambiguous. Same concepts can be expressed in many different ways. In order to make it possible to analyze the documents, the text needs to be first preprocessed and transformed into a structured format, which correctly represents the text content.

A lot of research has been done in the field of text mining recently and the results have been successfully applied in many areas like marketing, industry, and medicine. The next sections introduce a general theory of text mining along with the main techniques behind it.

3.1 Typical Steps of a Text Mining Process

We will start by describing some most common steps that are typically involved in a text mining process.

Information Gathering

This step involves understanding the target domain, identification of its key concepts, and general goals setting. It is important to have an idea of how the input data look like and what interesting information could be eventually extracted by their analysis. This step next involves careful consideration of information sources that sufficiently cover the application domain. The sources need to provide a sufficient amount of documents, and other necessary information. Implementation of mechanisms for collection of the documents from the individual sources may be often also required. If the documents come in different formats, like RDF, PDF, HTML, etc., they need to be converted into plain text form. If the information is structured or semi-structured,

it might be also necessary to unify the structure before further processing.

The way of storing the documents needs to be also considered. In many cases it is possible to simply save their contents as text files or XML files. However, for purposes of many tasks it might be better to save the documents in a database. There are many kinds of databases available and the choice of the proper one should be determined by the particular goals and the type of information to be stored.

More details on information gathering can be found in Chapter 4, which is dedicated to this topic.

Language Processing

Common learning algorithms cannot directly process the documents in their original form. The goal of language processing is to create a representation of the input documents that can be further passed as the input to various analytical algorithms. The next subsections describe some of the most common sub-steps of language analysis. Some of them may require specialized language analyzers for the particular language of the target documents. The proper choice of sub-steps usually depends on the particular needs of the subsequent analysis.

Tokenization

This process involves splitting the text into small meaningful sequences of characters called tokens. Tokens are usually words, numbers, punctuations, and various symbols. For European languages the tokenization is normally based mainly on separation by white-space characters. However, in case of other languages, such as Chinese or Japanese, it can be a much more complicated task.

Segmentation

Segmentation is similar to tokenization, but operates on a higher level. It splits the text into meaningful blocks like sentences, paragraphs, etc. The main challenge when identifying sentence boundaries in an English text is distinguishing between a dot that signals the end of a sentence and a dot that is a part of a word, like Mr., Dr., and so on. Tokenization and segmentation are often referred to as one process.

Part-of-Speech Tagging

Part-of-speech tagging is a process of adding so called *POS tags* to the words in the text, based on the context in which they appear. POS tags divide words into categories based on the role they play in the sentence. They provide information about the semantic content of a word. The most common POS tags are Article, Noun, Verb, Adjective, Preposition, Number, and Proper Noun. A detailed list of commonly used tags together with their customary abbreviations and descriptions can be found in [TAGS11].

POS tagging can be rule-based, in which case it is usually relatively fast. However, better precision can be often achieved by using the techniques of machine learning. These techniques naturally require that there is a training set available. The best trained taggers achieve an accuracy of about 95-98%.

Morphological Analysis

This process is sometimes considered to be a part of the part-of-speech tagging. Its main purpose is to identify word lemmas using the POS tags. A word lemma is the cannonical form of the word. For example words *think, thinks, thinking, and thought* have the same lemma which is the word *think*. Using the lemmas instead of the words themselves can simplify the subsequent analysis by reducing the dimension of the text representation.

Document Representation

The final step of the pre-processing phase is a choice of document representation that best serves the purposes of the subsequent analysis. Typically the documents are represented as vectors in a feature space. The features can be picked manually, but the most common approach is what is called a *bag-of-words model*, in which the features are simply determined by all the different words contained in all the documents. There are various ways how to assign weights to these features. Probably the simplest way is to assign 1 to the feature if the word represented by this feature is contained in the document and 0 otherwise. Another way is to use the frequency of the word in the document as the weight. A common approach is also to calculate the weight based on a representative set of documents using the following formula:

$$Weight = WordFrequency * \log\left(\frac{N}{DocFrequency}\right)$$

The *WordFrequency* is the frequency of the word in the document, *N* is the number of documents, and *DocFrequency* is the number of the documents that contain the word. The usage of the logarithm is based on the Zipf's empirical law that states

"*The most frequent word will occur approximately twice as often as the second most frequent word, which occurs twice as often as the fourth most frequent word, etc.*"
[KROH11]

Dimension Reduction

The main issue with the bag-of-words model is that the dimension of the feature space can grow very large. Additionally, most of the words contained in the documents usually have a very little informative value from the perspective of the target text mining task. These words not only unnecessarily increase the dimension, but can also add undesired noise into the analysis. There are various methods for identification and elimination of these words. These methods can be used individually or in combination with each other.

- **Stemming:** Many text mining tasks do not need to distinguish between various forms of a word, because the presence of the word itself has a sufficient informative value. Therefore the results of the morphological analysis can be used to reduce the dimension by representing the words by their lemmas.
- **Removing stop words:** Each language has some common words like articles, conjunctions, prepositions, etc., that usually do not contribute to the semantics of the documents. These words are usually referred to as *stop*

words. The appropriate set of stop words may depend on the particular task. Removing these words from the representation can also reduce the dimension.

- **Using nouns:** The main part of the semantics is usually carried by nouns. It is therefore in some cases possible to use the results of part-of-speech tagging and represent the documents only by the nouns they contain.

Once the proper document representation is created, it can be used as an input for various text mining tasks. Some of these tasks are described in the following sections.

3.2 Information Retrieval

Information retrieval is a process of finding documents that contain the answer to whatever the user is interested in regarding the target domain. An efficient information retrieval system needs to provide a query language that is expressive enough to sufficiently enable the user to specify attributes of documents he is interested in. The search can be performed over structured or unstructured data. In both cases, the data need to be properly indexed to enable quick response time. The retrieved documents are often ranked and ordered by the ranking. The rank determines how relevant the documents are to the users query. The quality of an information retrieval system is usually measured by two metrics:

$$Precision = \frac{(Number\ of\ retrieved\ relevant\ documents)}{(Number\ of\ all\ retrieved\ documents)}$$

If the precision is low, the user potentially wastes time by reviewing non-relevant documents.

$$Recall = \frac{(Number\ of\ retrieved\ relevant\ documents)}{(Number\ of\ possible\ relevant\ documents)}$$

If the recall is low, the user potentially misses important or interesting documents.

Information retrieval is not necessarily related only to text mining. However, there are certain text mining techniques that can help improving the search metrics.

3.3 Classification

Classification is also commonly referred to as *categorization*. The task is to classify a given instance of text into a pre-specified set of categories. There are two main approaches to this task:

- **Knowledge engineering:** The expert's knowledge about the categories is directly encoded into the system. The main drawback of this approach is the requirement of highly skilled experts for creation and maintenance of the knowledge-encoding rules.
- **Machine learning:** A classifier is built automatically by learning the properties of categories from a set of pre-classified training documents. The

process requires creation of a set of manually classified training instances. Although this might represent a labor intensive task, it is often much less complicated than the creation of the expert rules.

Formally, text classification can be defined as the task of approximating an unknown category assignment function $F : D \times C \rightarrow \{0, 1\}$, where D is the set of all possible documents and C is the set of predefined categories. The value of $F(D, C)$ is 1 if the document D belongs to the category C, and 0 otherwise.

The machine learning approach is in this case referred to as *supervised learning*, because it is guided by the set of training documents.

3.4 Clustering

Clustering, on the other hand, is an unsupervised learning process which separates the input collection of documents into so called clusters. The task is to group the given unlabeled collection into meaningful clusters without any prior information. A cluster should contain documents that are more similar to each other than to the documents in the other clusters.

However, an important thing to be kept in mind is that there are many ways in which documents can be similar. The similarity function is usually based on the distance between the feature vectors in some metric, such as *Euclidean metric*, *Chebyshev metric*, and so on.

One application of clustering is in the domain of information retrieval. A common issue of standard information retrieval systems is that the same concepts are often expressed by different terms in different texts. Clustering, which is based on overall similarity, may help improve the recall of the search by returning the whole cluster. The idea is based on an assumption called *cluster hypothesis* that states

"*Relevant documents tend to be more closely related to each other than to non-relevant document.*" [KROH11]

Best precision can be achieved if the clustering is done during the search and directly depends on the particular user query. However, this approach can naturally slow down the search process.

3.5 Information Extraction

Information extraction refers to the automatic extraction of structured information from an unstructured text. This includes extraction of entities, their attributes, and relationships between them. The data can be directly presented to the user, or may be stored in a database and used for indexing purposes in information retrieval systems. Information extraction involves a lot of sub-domains, including:

- **Named entity recognition:** Identification of all mentions of proper names and quantities in the text. These include people names, geographic locations, organizations, dates, times, monetary amounts, percentages, and so on.
- **Coreference resolution:** Identification of identical entities in the text that are

represented by different words. For example a car can be in the text represented by the word *car* and later by the word *it*, referring to the same car.

- **Relationships extraction:** Identification of relationships among entities. A relationship can be for example between the entity A - a medicine, and the entity B - a disease, where the medicine A cures the disease B.
- **Summarization**: Generation of summaries from long text documents.

Similar to classification, there are two major approaches to information extraction: **knowledge engineering** and **machine learning**. More information on the topic can be found in [FESA07], [APIS99], [CUNN04], and [SARA08].

3.6 Evaluation of results

There are various methods that can be used for each of the described tasks. The effectiveness of these individual methods often depends on many factors and it is usually required to compare multiple methods before choosing the one that is the most suitable for the particular task. In order to make the comparison possible, the methods need to go through an evaluation process. The purpose of this process is to describe the quality of the methods in numbers, so the more suitable methods can be clearly distinguished from the less suitable ones.

A common approach in case of the classification task is to divide a set of classified documents from the target domain into a training set and a testing set. The classifier is trained on the set of training documents and subsequently applied to the testing set. It is important not to use the testing set in any way during the classifier training. The classifications done by the classifier are then compared with the original categories of the documents in the testing set. The numbers of correct and incorrect classifications can be then used to calculate various metrics which determine the quality of the classifier.

3.7 Text mining algorithms

This section describes some of the well-known algorithms used in text mining. Although they are described here from the perspective of text classification, many of them can be adapted for the other tasks as well.

Decision Trees

A *decision tree* classifier is a tree in which the internal nodes are labeled by the selected features of target documents and the leaves are labeled by categories. The algorithm classifies a document by starting at the root of the tree and moving successively downward via the branches whose conditions are satisfied by the document until it reaches a leaf node.

The tree is built recursively by picking a feature *f* at each step and using it to divide the training set. Typical criteria when choosing the feature are information gain and entropy. However, the trees generated in such a way are prone to overfit the training set and their performance is usually considered to be inferior to other classifiers. The great advantage of this algorithm is that its basics are easy to

understand and it is often used as a baseline for comparison with other algorithms. More information about the algorithm can be found in [HAKA06].

Neural networks

Neural networks are among the best known algorithms when it comes to machine learning. The simplest type of neural network is a perceptron, which is essentially a linear classifier specified by a vector w and a number b. Given an input document represented by a feature vector x, the perceptron calculates an output function $f = w * x + b$, where * represents a scalar product. If the value is greater than zero, then the document belongs to category A, otherwise the document belongs to category B. If a misclassification occurs, the parameters w and b are adapted to correctly split the vector space.

Nonlinear networks contain one or more hidden layers between the input and output layers. The adaptation of the parameters is in such case performed using the algorithm called *back propagation*. However, the experiments have shown very small improvement of nonlinear networks over their linear counterparts in the text classification task [FESA07].

Naive Bayes Algorithm

Naive Bayes is a probabilistic classifier based on the Bayes Theorem:

$$P(C|D) = \frac{P(D|C) * P(C)}{P(D)}$$

The algorithm assigns documents to categories based on calculated probabilities. It is called naive because it assumes that a presence of a term in a document is statistically independent on the presences of the other terms in that document.

Given a training set, it is not possible to calculate the probability that a document D belongs to a category C directly. However, it is possible to calculate the probability of the category C and the probability that a term T is present in a document which belongs to the category C.

Or in other words, it is not possible to calculate $P(C|D)$ directly, but it is possible to calculate $P(C)$ based on the number of documents that belong to the category, and $P(T|C)$ based on the frequency of the term T in the category.

Using the assumption mentioned above, it is possible to compute $P(D|C) = \prod P(T_i|C)$ and then use the Bayes Theorem to calculate $P(C|D)$. $P(D)$ may take any value other than zero because it remains the same for all the categories.

K-Nearest Neighbor

K-nearest neighbor (*KNN*) does not build explicit declarative representations of categories but rather stores the representations of the training documents together with their category labels. Each document can be represented by a feature vector, which corresponds to a point in Euclidean space. It is also possible to use a different metric. The similarity between documents in this space is determined by their distance. To classify a document D, the algorithm checks the categories of the k training documents that are the most similar to D and typically chooses the category that is the most common among the documents. The documents can be also given different weights based on their distance from D. The question remains how to

choose the proper value of *k*. This is usually done empirically. However, various experiments have shown that the best effectiveness is usually achieved by choosing a value between 20 and 45 [LACR96], [YANG01].

Support Vector Machine

Support Vector Machine (*SVM*) classifier in its binary form can be considered to be a hyperplane in the feature space that separates the points that represent the positive instances of the category from the points that represent the negative instances. The classifying hyperplane is chosen during training as the unique hyperplane that separates the known positive instances from the known negative instances with the maximal margin. The margin is the distance from the hyperplane to the nearest point from the positive and negative sets. It happens sometimes that the sets are not linearly separable, in which case the original finite dimensional space needs to be mapped into a much higher dimensional space. The SVM hyperplane is often fully determined by a relatively small subset of training data, which are called support vectors, while the rest of the training data have no influence on the trained classifier at all. More information about the algorithm can be found in [CRSH00].

Hidden Markov Model

Hidden Markov model refers to a type of probabilistic model based on a sequence of events. The events can be for example sequential consideration of the words in a text. It is presumed that the individual events are parts of some larger constituents, like names of particular type. Whether a word is part of a name or not is a random event with an estimable probability. This probability can be estimated from a training set.

It is also presumed that there is an underlying finite state machine that changes state with each input element. The probability of a recognized constituent is conditioned not only by the words seen, but also by the state that the machine is in at the moment. Constructing an HMM classifier consists of construction of a state model and subsequent examination of sufficiently large training set to accurately estimate the probabilities of the various state transitions.

More information about the algorithm can be found in [FESA07].

4 Analysis

According to [QWJS09] the number of articles about reported software vulnerabilities reaches several hundred a week. This might not seem as a very high number, but reading all of the articles definitely consumes a considerable amount of time. Additionally, the number has increased rapidly over the last several years and can be expected to grow further. The reason is not only the larger number of software products that are available, but also the increased awareness of the importance of software security. There are almost fifty thousand software vulnerabilities recognized by CVE and most of them have been discovered during the last decade. The main issue which comes along with the articles is that usually only some of them are truly relevant for a certain person or a particular information system. The domain of relevance can be additionally determined by the profession and interests of the person. He can be a computer network administrator, a security researcher, or just a regular person interested in computer security.

The main goal of this analysis is to provide means for efficient filtering and information retrieval in the domain of articles about software vulnerabilities. We will start by introducing the Lucene library, which was used to implement some of the advanced search techniques, and the GATE text mining framework, which was the primary tool used for the advanced parts of this analysis.

4.1 Lucene

Apache Lucene is a search engine library written entirely in Java. It provides means to index text documents and perform search over them in many various ways. It has probably one of the best full-text search supports among similar open source projects and can also compete with many commercial ones. Lucene is distributed under the Apache License, which makes it available in both commercial and open source sphere.

Many relational database engines come with some full-text search support of their own. This usually includes searching by wildcards using a LIKE statement. However, this search usually does not use any kind of full-text index, which in many cases results into a full table scan. Another common feature is searching by keywords.

Lucene supports wildcards, keywords, and much more. It usually achieves very good performance thanks to special full-text indexes. But the library is not a database engine. It can be used as a document oriented storage in case of some simple applications. However, it works probably best in combination with a fully-featured database.

More information about Apache Lucene can be found in [LUCE11].

Lucene Search Functionality

The following subsections describe some of the full-text search functionality provided by Lucene.

Keywords Search

Lucene can be configured to perform tokenization during the indexing, which splits the input text into words and other tokens. Besides traditional spaces and other white-space characters, the set of token separators also includes characters like '_', '-', '.', etc. This makes it possible to efficiently search for documents that contain a specific set of tokens.

The engine that performs the tokenization is called *Lucene analyzer*. In addition to the traditional Lucene analyzer, there is also a special engine called *Snowball analyzer*, which can be applied to English texts. It differs in that the tokens are converted into their lemmatized form before they are indexed. This can in some situations simplify the search, because the user does not need to worry about various possible word forms.

In addition to search by individual tokens, it is also possible to look for whole phrases. Another type of search can find documents, where one word is in proximity of some other word. The Lucene key words search functionality is very rich and it is possible to identify some aspects of text mining in it. There is for example a prototype functionality, which can be used to find documents that are similar to some other document. This is done by eliminating stop words and using the rest of the words to determine the similarity.

Fuzzy Search

The way how the fuzzy search works is that it looks for similar words rather than an exact match. This means that for example searching for a word *vulnerable* will also retrieve documents containing word *vulnerably* or mistyped words like *vulnerablr*. A threshold needs to be specified, which determines the level of deviation that is allowed.

The search is based on the *Levenshtein distance algorithm*. The algorithm calculates a distance between two strings, which is defined as the minimum number of operations needed to transform one string into the other. The allowed operations are insertion, deletion, and substitution of a single character. The calculation of the distance is an example of dynamic programming. The algorithm incrementally calculates the distances between prefixes of the two specified strings by flood filling a matrix of size MxN, where M is the length of the first string and N is the length of the second string. More information can be found in [WILD10].

The algorithm has been successfully applied in spell checking, speech recognition, DNA analysis, and other areas.

Wildcard Search

This kind of search allows searching for parts of a word by masking its characters using symbols '*' and '?', also known as wildcard symbols. To mask only the suffix of a word, it is recommended to use the *prefix search*, which is optimized for this task.

The library is also capable of searching by regular expressions. However, this functionality needs to be used carefully because of potential slower performance.

Scoring the Results

If the number of results for a given search statement is very high, it is very likely that the user would only want to go through those few of them that best meet the search

criteria. One of the powerful features that Lucene comes with is the ability to score the results. The relevance score associated with a result determines how closely this result satisfies the search statement. The user can affect the result score by boosting various parts of the statement and increasing their importance this way. The scoring formula is rather complex, but it is basically determined by the number of found matching words and the precision of the match in case of the fuzzy search, the wildcard search, and so on. It is also possible to adapt the formula itself, but this is only recommended to advanced users.

4.2 GATE

"*GATE is an infrastructure for developing and deploying software components that process human language.*" [CUNN11]

GATE is open source free software developed and maintained by the GATE community. The software ties together various techniques that are used as part of natural language processing. These include linguistic techniques, machine learning, and so on. Several popular projects from this domain and related domains, like LingPipe, OpenNLP, Weka, WordNet, etc., are also integrated with GATE. One of the challenges that GATE is trying to deal with is providing the unified environment for working with these various projects. The two major parts of the software are:

- **The GATE Developer:** An IDE, where various components can be configured and combined using a graphical interface.
- **The GATE Embedded:** A framework, which can be integrated directly into Java code. Everything that can be done using the GATE Developer can be also done using the GATE Embedded.

More information about GATE can be found in [GATE11] and in [CUNN11].

Considered Alternatives

GATE was chosen because it has been already developed over fifteen years and there are still many people that are actively involved in its development. It has also been already used in couple of successful projects and its mailing-list provides active support to its users. The following subsections describe some other options that were also considered.

Mahout

Apache Mahout is a machine learning library written in Java and implemented on top of Apache Hadoop. In spite of being relatively young, it already supports many popular data mining and text mining techniques and algorithms. The library is distributed under a commercially friendly Apache Software license. More information about the project can be found in [MAHO11].

Weka

Weka is also a machine learning library written in Java. It contains tools for data preprocessing, classification, regression, clustering, association rules, and visualization. It is distributed under the GNU General Public License. A large part of the Weka functionality has been already integrated with GATE. More information about the library can be found in [WEKA11].

Working with GATE

As mentioned above, there are two major ways how to work with GATE. It can be integrated directly into Java code by including the GATE Embedded library, registering desired plugins, creating appropriate objects, and calling appropriate methods. However, it is recommended to start by using the GATE Developer to learn about the main aspects and ideas that come with this software.

GATE Language Resources

GATE works primarily with plain text documents. Many other formats can be also imported and converted into their plain text form. To keep them organized, it is possible to create a so called *corpus*, which is a collection of selected documents. The GATE documents and the GATE corpora are commonly referred to as *language resources*.

A document can have a set of features and a list of annotations. The features are key-value pairs which are the attributes of the document. The annotations on the other hand give a specific meaning to some particular parts of the document content. An annotation can be over a single word, or over a sequence of words. It can also start or end in the middle of a word, but this is not a very common case. Each annotation is described by a start index, an end index, a type, and a list of its features. Each annotation type has a name and a color which is used to display the annotations of this type.

To keep the annotation types organized, they are put into annotation sets. Each annotation type can be only inside one set. The name of the set can be arbitrary, but there are two special annotation sets, which is good to be aware of:

- **The default annotation set:** This annotation set is always present and its name is an empty string.
- **Original markups:** If there are annotations present inside the original document and they can be deduced during the import, they are put into this annotation set. For example if the original document is an XML file, each XML tag is replaced by an annotation. The name of the annotation and its features correspond to the name and the attributes of the replaced XML tag.

The annotations can be also added to the documents manually by selecting parts of the text, assigning an annotation type, and adding annotation features. This task can be very labor intensive and usually requires a good knowledge about the target domain. The annotation types can be also organized in a form of ontology, which besides other advantages, allows the user to perform annotating more efficiently. The ontology is a language resource and can be loaded into GATE from

various formats.

GATE Processing Resources

The last way how to add annotations is by automated processing. There is a large number of plugins that can be easily loaded into GATE. Each of the plugins provides a set of so called *processing resources*. Each processing resource is designed to perform some specific task. For example *tokenizer* splits the text into individual words, numbers, and punctuations. It does this by adding an annotation for each word, number, punctuation, and space that is present inside the text. It also adds features to the annotations that provide some additional information, such as whether the word starts with a capital letter. The processing resources can be organized into a *pipeline*, which can be executed over a single document or over a whole corpus.

The GATE Datastore

The documents and corpora can be stored inside a *datastore* and later retrieved for further processing. It is possible to retrieve only a corpus and run a pipeline over it. The documents contained in that corpus are then retrieved, processed, and saved back to the datastore one by one. This can significantly reduce the amount of memory that is needed for the processing.

It is also possible to create a special type of datastore called *Lucene-based datastore*. This datastore uses Lucene to index the stored documents and can be queried using a special syntax.

GATE Plugins

GATE itself is an environment designed for comfortable development. The major part of its processing power is present inside its plugins. There are plenty of plugins that are directly available and even more can be downloaded from the GATE website. Besides containing the processing resources, the plugins also sometimes provide so called *visual resources*. These are additional graphical components, which can be used by a user to perform specific tasks.

The following is a list of the most important plugins. Most of them were used during the analysis described in this chapter.

ANNIE

The letters in *ANNIE* stand for *A Nearly New Information Extraction System*. It is a collection of resources that are primarily designed for language processing. The resources include the *English tokenizer*, the *sentence splitter*, the *part-of-speech tagger*, the g*azetteer*, the *semantic tagger,* and some others.

The gazetteer is created from one or more lists of words. The lists have special syntax and each of the words can be assigned to a major type and a minor type. The gazetteer compiles the lists into a finite state machine. Any text tokens that are matched by the machine are annotated with features specifying the major type and the minor type. The gazetteer can be additionally configured using several options like case sensitivity, character encoding, etc.

Although the traditional gazetteer is usually sufficiently fast, it might be sometimes more efficient to use so called *hash gazetteer.* This type of gazetteer uses hash maps rather than a finite state machine and has proven to be faster in many

situations.

The semantic tagger is based on a special language called *JAPE*, which provides means to match and manipulate already existing annotations based on the specified grammar. It is also referred to as the *JAPE Transducer*. The matching is done based on rules that are similar to regular expressions. If the text was previously analyzed using the part-of-speech tagger, it is for instance possible to match some specific noun together with its preceding adjectives to match a phrase which might have some specific meaning. The matched pattern can be then manipulated in various ways. It is possible to add a new annotation or add features to the matched annotations. Advanced users can also write Java code directly to manipulate the matched content. More examples on the usage of the JAPE grammars can be found later in this text.

There are also several other plugins, like LingPipe and OpenNLP, which provide a similar functionality to ANNIE. Although ANNIE usually works quite well on common text, it is sometimes worth consideration to use these plugins as well.

GATE Tools

From the perspective of our analysis, probably the most interesting tool contained in this plugin is the *GATE morphological analyzer*. This processing resource identifies the lemma and the affix of each tokenized word present in the text. These values are then added as features to the token annotations. The text needs to be first processed using the tokenizer, the sentence splitter, and the part-of-speech tagger in order to make it possible to apply the GATE morphological analyzer to it.

Ontology Plugin

The *ontology plugin* provides the ontology support. An ontology can be used for manual annotation, but can also be integrated into the semantic tagger if the text is annotated based on the ontology. The tagger is then aware of the relationships described by the ontology, which makes it for instance possible to match all children of an entity.

The plugin works best in combination with the *ontology tools plugin*, which provides various visual resources for working with ontologies. These include an ontology editor and tools for comfortable ontology based annotation.

There is also a type of gazetteer called *OntoGazetteer* which can be provided an ontology and a mapping file in addition to the standard word lists. The mapping associates the words from the lists with the ontology classes. The OntoGazetteer assigns these classes rather than major or minor types.

Learning Plugin

This plugin provides the machine learning support. It contains only a single processing resource called *batch learning*. This processing resource needs to be first configured using a configuration file, which describes the algorithm and the various aspects that are supposed to be used for the learning process. The tool can be used in a learning mode, an application mode, or an evaluation mode. There are five algorithms that are currently supported: the *Naive Bayes,* the *K-Nearest Neighbor*, the *C4.5 decision trees*, the *Perceptron Algorithm with Uneven Margins*, and the *Support Vector Machine*. The first three algorithms are provided by the Weka library. The

implementations of the other two come from the developers of GATE. Further details about the configuration and the usage will be described later in this text.

4.3 Basic Analysis

We will start by looking at the target articles from the human perspective and identifying their basic features. There is sometimes a lot of information that can be extracted simply by human observation. The first thing to notice is that the articles from the same source usually follow the same pattern. Many sources even provide the articles in a partially structured form. Each source has its own structure, but the information usually includes some kind of title, a release date, a description, a list of affected products, and a list of references to other sources.

Filtering by Release Date

One of the ways how the articles can be filtered is by the release date. If they are newer, they are more likely to describe a vulnerability that has not been patched yet and therefore requires more attention. If a person reads the articles regularly, he is usually more interested in the new articles than the old ones. It is also very likely that the articles that describe the same vulnerability are released approximately around the same date, shortly after the information about the existence of the vulnerability was disclosed.

Filtering by Affected Products

Another way of filtering might use the affected products. This is probably one of the most natural ways how to identify the articles that are relevant for an administrator of an information system. A computer can be from the perspective of software security described by the operating system it is running and by a list of installed software products. The same counts for an information system, but in a larger scale.

The exploitability of vulnerability can of course also rely on some other aspects, like the configuration of the affected products, or the underlying hardware. However, ignoring such aspects usually introduces only a little risk of false positive results, which is in most cases negligible.

Unfortunately, there is one significant complication that comes along with this kind of filtering. Each source has its own unique way of assigning names to affected software products. Since the naming does not follow any well-established standard, the variety in the naming can be significant. The only source that tries to follow the CPE standard is NVD. And even this standard opens some space for variety. For example *Microsoft Windows 2000* is in the CPE database listed twice: once with the vendor name *Microsoft*, product name W*indows* and version *2000*, and once with the vendor name *Microsoft* and the product name W*indows_2000*. This makes it hard to predict the names that will be used as references to the affected software products.

The sources usually provide some list of all the affected products listed in their articles. The issue with these lists is that they do not usually help to keep track of the new articles, because the first time when a specific product appears in the list is usually the time when an article about the product vulnerability is published.

A way how to approach this might be not to try to filter by the exact product names, but rather look for some key tokens. The order of the tokens in this case

usually does not matter. It might be a good idea to also make the search case insensitive and include characters like '_' and '-' into the list of token separators. Allowing some fuzziness can help dealing with mistypings and little variations among terms that represent the same thing. Using wildcards can also help dealing with certain types of shortcuts.

A bit more complicated are shortcuts like *IE* for the *Internet Explorer*. It is possible to make an assumption that the words that are very short might represent a shortcut, and use the individual letters for wildcard search. However, this approach can considerably increase the chance of false positive results.

Even more complicated are shortcuts like *FF* for the *Mozilla Firefox* or *MS* for *Microsoft*. Dealing with all possible variations would probably require a much deeper analysis and could probably be a topic for a separate research.

Filtering by References

Another way of filtering could use the references to other sources. These can be used to identify related articles, like for example the articles that describe the same vulnerability. The variability of strings representing the same reference is fortunately much lower than the variability in case of the affected products. A common type of reference is a CVE number, which is an identifier of a vulnerability described by CVE. A little bit of variability comes along with URL references. The URLs might or might not contain the specification of the protocol at the beginning, and some URLs can be written both with and without the *www.* string. However, this kind of variability is much less complex and can be usually simply dealt with using the wildcards.

Filtering by Description

The last approach that we introduce will be filtering by the description. The description is in most cases only a plain text with no strict structure. Analyzing the description is very similar to analyzing blogs and mailing-lists, so the techniques that we apply here are applicable to them as well.

One way how to look at the description is that it is a set of words. There is a good chance that if the text describes some kind of vulnerability, it also contains some particular words that represent that kind of vulnerability. The name of the affected product and other information is also commonly mentioned in the description. So the basic approach here could be again looking for the key tokens. The user needs to know what to look for to get the right results. Allowing some fuzziness and wildcards might also have a meaning here. The search could be also applied to the title of the article, or some other text attributes.

The next sections describe some of the ways how the description text can be analyzed using the text mining techniques.

4.4 Classification

One of the most common text mining tasks is classification. Distribution of documents into categories can help in organizing the documents, searching for related documents based on the common category, and so on.

This section focuses on describing how the articles about software

vulnerabilities can be classified based on the Common Weakness Enumeration standard. The goal of this analysis was to identify CWE categories of individual documents based on their description text. This could for example allow the users that are interested in specific areas to filter the articles by the type of vulnerability that they describe.

The CWE standard is mainly followed by the National Vulnerability Database. It is very rare that any other source would classify the articles based on this standard as well. The articles from NVD were therefore used as the training set, so the resulting statistical model could be applied to classify the articles from the other sources.

Not all the articles from NVD contain the CWE category element. It was necessary to select those documents that were classified, extract their description texts, and store the descriptions together with their categories. A simple programming solution for this task can be found among the source codes on the enclosed CD, see Appendix F. The application stores the information from each document in a separate file in the following format:

```
<Text class="{CWE_category_id}">{The description text}</Text>
```

The application was run on 26th of May 2011 and extracted 18 723 classified texts from all the NVD articles that were published prior to this date. Because the resulting files were in a simple XML format, they could be directly loaded into GATE and were automatically converted into annotated description texts.

Classification Settings

There are several machine learning algorithms supported by GATE. The various algorithms can provide various results depending on their configuration and on the specifics of the task for which they are used. The goal of this analysis was to compare multiple algorithms and their various configurations and choose the most appropriate classifier for the CWE classification task.

The algorithms chosen for this experiment were the K-Nearest Neighbor (KNN), the Perceptron Algorithm with Uneven Margins (PAUM), and the Support Vector Machine (SVM). For each algorithm were chosen two representative configurations. The KNN algorithm was configured once with k = 20 and once with k = 40.

The choice of the configurations for PAUM was inspired by the research described in [LZHT02]. The negative margin and the positive margin were set in the first version to values -1 and 1, and in the second version to values 1 and 50. The learning rate was set for both versions to 0.3.

The choice of parameters for SVM was inspired by suggestions from [CUNN11]. The type of the kernel for the first version was set to linear, while the type of the kernel for the second version was set to cubic (polynomial with the degree set to 3). The cost associated with training errors and the margin coefficient were in both versions set to values 0.7 and 0.4.

Further details on the algorithms and their configuration can be found in [LZHT02] and [CUNN11].

The classifiers were configured to run in an evaluation mode, in which the

evaluation was done by randomly splitting the input documents set into a training set and a testing set with a splitting ratio of 4:1. The exact number of training documents was 14978 and the number of testing documents was 3745.

To achieve a more reliable comparison, the process was configured to perform each evaluation in three separate runs and the average values from these runs were taken as the results.

The comparison was afterward done using the following metrics calculated from the number of the correct classifications, the number of the incorrect classifications, and the number of all testing documents:

$$Precision = \frac{Number\ of\ correct}{Number\ of\ correct + Number\ of\ incorrect}$$

$$Recall = \frac{Number\ of\ correct}{Number\ of\ all}$$

$$F-measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

The *F-measure* represents a general score used to determine the accuracy of a classifier. This formula expects that the precision and the recall are treated equally. However, if the target system required different weights to be assigned to these two metrics, the formula would need to be adapted to meet the requirements.

There are certainly also other metrics that could be used for the comparison, like the overall evaluation duration, the memory requirements, and so on. These metrics, however, were not considered to be relevant for the analysis. The overall evaluation duration was in all cases approximately 2.5 to 4 hours.

GATE also provides a probability with every classification that represents the confidence with which the classification was done. The calculation of the probability is provided for all the supported algorithms. It is possible to specify a probability threshold. Any classification with a probability lower than this threshold is omitted. If not stated otherwise, the following experiments were done with the threshold set to 0.5.

Basic Classification

The first experiment used the words contained in the documents in their original form. The only preprocessing that had been done was tokenization of the document texts using the ANNIE English tokenizer. All the configurations of the algorithms were done the way they were described above. Table 4.1 shows the average values measured for the individual algorithms during this experiment.

There are several things that we can read from the results presented in the table. Both PAUM and SVM showed better accuracy in this task, SVM being slightly better in this case. Different configurations of the algorithms had different impacts on the results. It is interesting to notice that PAUM (1,50), unlike the other algorithms, classified all of the documents, although some of the documents were classified incorrectly. This indicates that all classifications done by this algorithm had a probability higher than 0.5.

Table 4.1: *The results measured during the evaluation of various algorithms and their configurations after the documents had been processed by the ANNIE English tokenizer.*

Algorithm	Correct	Incorrect	Precision	Recall	F-measure
PAUM (-1,1)	2884.00	324.67	0.8988	0.7700	0.8295
PAUM (1,50)	2945.00	800.00	0.7864	0.7864	0.7864
KNN (20)	2461.00	1269.00	0.6598	0.6571	0.6585
KNN (40)	2350.33	1377.00	0.6306	0.6276	0.6291
SVM (linear)	2953.67	356.00	0.8925	0.7887	0.8374
SVM (cubic)	3119.00	539.67	0.8525	0.8328	0.8426

The algorithm with the best F-measure was SVM(cubic). While having the best recall among the algorithms, it also showed slightly lower precision than its linear counterpart.

The results presented by GATE also revealed that the articles from NVD are classified into only nineteen categories out of the 870 categories that are recognized by CWE. A description of these categories can be found in Appendix A.

Stemming

The input documents were in the next experiment further processed by the ANNIE sentence splitter, the ANNIE part-of-speech tagger, and the GATE morphological analyzer. This time the algorithms were configured to learn from the lemmas of the words with the goal to reduce the dimension of the feature space. All the other settings remained unchanged. Table 4.2 shows the values measured during the evaluation.

Table 4.2: *The results measured during the evaluation of various algorithms and their configurations after the documents had been further processed by the ANNIE sentence splitter, the ANNIE part-of-speech tagger, and the GATE morphological analyzer.*

Algorithm	Correct	Incorrect	Precision	Recall	F-measure
PAUM (-1,1)	2930.33	370.33	0.8878	0.7825	0.8318
PAUM (1,50)	2945.33	799.67	0.7865	0.7865	0.7865
KNN (20)	2665.33	1048.00	0.7178	0.7117	0.7147
KNN (40)	2634.33	1062.33	0.7126	0.7034	0.7080
SVM (linear)	2934.67	350.33	0.8934	0.7836	0.8345
SVM (cubic)	3126.00	555.00	0.8492	0.8347	0.8419

If we compare the results with Table 4.1, we can see that while the modification had a significantly positive effect on the KNN algorithm, it had almost

no effect on the other algorithms. Considering the amount of time needed for preprocessing of the documents, it turns out to be wiser not to perform this sub-step if the PAUM algorithm or the SVM algorithm is to be used for this task. However, we also need to take into account that words in many languages, like the Czech language and the Slovak language, typically have more forms than English words. Applying stemming to texts written in these languages could give different results.

Removing Stop Words

The goal of this experiment was to test the effect of removing various common words, numbers, punctuations, and symbols from the set of tokens used for the learning process. A list of the stop words that were used for this task can be found in Appendix B. The standard ANNIE gazetteer was configured to use this list and applied to the documents. The gazetteer added an annotation of type *Lookup* for every stop word found in the documents. Figure 4.3 shows the JAPE grammar that was subsequently applied to filter the tokens used for machine learning, keeping only the words that did not belong to the set of stop words.

Figure 4.3: *The JAPE grammar used to filter the stop words out of the input documents. The grammar also filters numbers, punctuations, and symbols.*

```
Phase: StopWordsRemoval
Input: Token Lookup

Rule:StopWordsRemoval
Priority:1
({Token within Lookup}
):stopWord
-->  { inputAS.removeAll(bindings.get("stopWord")); }

Rule:KeepOnlyWords
Priority:2
({Token.kind != "word"}
):notWord
-->  { inputAS.removeAll(bindings.get("notWord")); }
```

Table 4.4 shows the values measured during the subsequent evaluation.

Table 4.4: *The results measured during the evaluation of various algorithms and their configurations after removing stop words from the documents.*

Algorithm	Correct	Incorrect	Precision	Recall	F-measure
PAUM (-1,1)	2884.67	331.00	0.8971	0.7703	0.8288
PAUM (1,50)	2942.67	802.33	0.7858	0.7858	0.7858
KNN (20)	2596.33	1142.33	0.6945	0.6933	0.6939
KNN (40)	2579.00	1160.00	0.6898	0.6887	0.6892
SVM (linear)	2980.00	388.00	0.8848	0.7957	0.8379
SVM (cubic)	3128.67	554.33	0.8495	0.8354	0.8424

Neither PAUM nor SVM showed any decisive improvement or degradation. Interesting is the little degradation of results in case of KNN. We can assume that this was caused by removal of some important tokens. These could be some numbers, symbols, or words that were incorrectly identified as stop words. This indicates that even some common words and non-word tokens can sometimes participate on the semantics of the documents. It is therefore important that the set of tokens used for the filtering is chosen carefully.

Reducing the Number of Categories

Considering that the CWE database lists 870 categories, it might be surprising that only nineteen of them are used to classify the articles from NVD. However, if the number was significantly higher, it would be worth consideration to reduce it and possibly get better classification results. The reduction could be done by merging various categories together. Because the CWE database is organized in a form of taxonomy, the categories could be replaced by their super-categories.

The categories in this experiment were replaced by six top level categories that cover all the nineteen sub-categories found in articles from NVD. The CWE database was transformed into an OWL ontology in RDF format [OWL11] using an XSLT transformation that can be found among the XML related files on the enclosed CD under the name `CWEtoOntology.xslt`. The classified NVD articles were then converted using a simple ontology aware JAPE grammar, which can also be found on the CD. The evaluation was then performed with standard settings.

However, instead of improvement, the modification resulted into significant degradation of the results. Analyzing the cause showed that all the articles in the testing set were classified with the same super-category. Further analysis showed that this super-category covered more than a half of all the NVD articles and was far superior to the other five super-categories. Assuming that this was the cause of so many misclassifications, optimizing the task would probably require choosing such super-categories that evenly cover the set of input documents. Considering the fact that the CWE database is organized in taxonomy and not just a simple tree structure, this might not be an easy task. Because the number of the categories was already relatively low, there was no need to continue with the experiment.

Increasing the Probability Threshold

The goal of this last experiment was to find out more about the probability distribution of the correct and the incorrect classifications done by the algorithms. The probability threshold was this time set to 0.8. The main focus was to observe the effect on the SVM algorithm, which showed the best results during all the experiments. Because the stemming and the stop words removal showed no distinctive improvement in case of this algorithm, the evaluation was done with the same set of documents and the same remaining settings as in the first experiment. Table 4.5 shows the measured values.

We can see that also in this case PAUM (1,50) classified all of the documents, meaning that all the classifications were done with a probability higher than 0.8. Interestingly, the number of correct classifications was slightly increased despite the increased probability threshold. Because the difference is very little, we can assume that it was caused by the random separation of documents into a training set and a

testing set.

Table 4.5: *The results measured during the evaluation of various algorithms and their configurations after increasing the probability threshold to 0.8.*

Algorithm	Correct	Incorrect	Precision	Recall	F-measure
PAUM (-1,1)	2662.67	184.33	0.9353	0.7110	0.8078
PAUM (1,50)	2953.00	792.00	0.7885	0.7885	0.7885
KNN (20)	2192.67	630.33	0.7767	0.5855	0.6677
KNN (40)	2082.00	740.00	0.7390	0.5560	0.6344
SVM (linear)	2532.33	105.33	0.9600	0.6762	0.7935
SVM (cubic)	2689.33	158.33	0.9444	0.7181	0.8159

In case of all the other algorithms the number of correct classifications naturally dropped, resulting into decreased recall. It is interesting, however, that the precision was in all the cases increased. This change is not a matter of course.

In mathematical terms, it can be described by the following inequality:

$$\frac{A-\alpha}{(A-\alpha)+(B-\beta)} > \frac{A}{A-B}$$

The left-hand site of the formula represents the precision measured with the probability threshold set to 0.8, and the right-hand site represents the precision measured with the probability threshold set to 0.5. This inequality can be then transformed into the following form.

$$\frac{A-\alpha}{A} > \frac{B-\beta}{B}$$

If we call the classifications done with a probability higher than 0.8 *high classifications*, and the classifications done with a probability higher than 0.5 *normal classifications*, then we can express the relation as follows

$$\frac{\#\,correct\ high\ classifications}{\#\,correct\ normal\ classifications} > \frac{\#\,incorrect\ high\ classifications}{\#\,incorrect\ normal\ classifications}$$

Or in other words, the correct classifications are more commonly done with a higher probability than the incorrect classifications.

Conclusion

Although the description texts in articles about software vulnerabilities are usually relatively short, they can be used to classify the articles into CWE categories. Various algorithms and various configurations can give different results. Some algorithms can have better precision and some can have better recall. The best achieved precision was 0.96 by SVM (linear), and the best achieved recall was 0.8354 by SVM (cubic). The final choice of the proper algorithm should depend on the

particular requirements.

It is often possible to improve the precision of the algorithms by increasing the probability threshold. This modification, however, usually leads to a degradation of recall, so it is important to choose the value for the probability threshold carefully.

Various language processing techniques can sometimes also be helpful. However, they need to be first tested and correctly adapted to the particular task. Various modifications can improve the results of one algorithm and degrade the results of another algorithm at the same time.

There was one algorithm that was showing slightly better F-measure than the other algorithms during all the experiments. Figure 4.6 shows the results of the evaluation of this algorithm in the last experiment. The complete configuration used for this algorithm can be found in Appendix C.

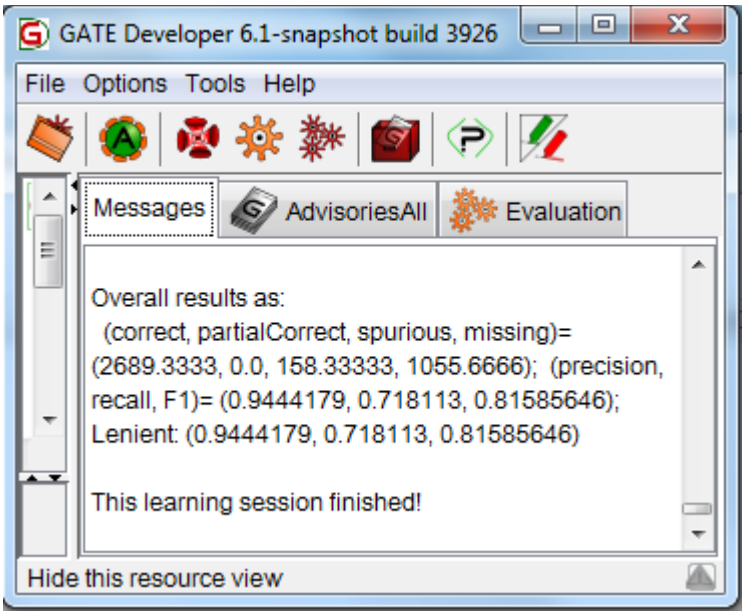


Figure 4.6: *Results of the evaluation of the SVM (cubic) algorithm presented by GATE in the last experiment*

4.5 Information Extraction

This section focuses on describing how the key information can be extracted from the articles about software vulnerabilities to simplify their reading and allow additional ways of filtering. To perform this task, it is first required to determine what the key information is.

The description text in the articles usually follows very similar pattern. This pattern might not be well-defined, but there are certain types of information, which commonly appear in the text. These types can be put into categories and possibly organized into a taxonomy. This taxonomy can be later turned into an ontology by specifying various relations between the categories.

The following taxonomy was designed for the purposes of this experiment. It is inspired by various standards like OVAL, CVE, and CPE.

- **Vulnerability Information**
  - **Vulnerability Details:** Various information about the vulnerability
    - **Cause:** Reason why the affected product is vulnerable. Typically this information describes the details of the software vulnerability itself.
    - **Complexity:** This information describes how difficult it is for an attacker to exploit the vulnerability. This typically includes distinguishing whether the vulnerability can be exploited remotely, or whether the attacker needs to first gain access to the information system.
    - **Impact:** Various possible impacts that a successful exploitation of the vulnerability can generally have on an information system.
    - **Type of Weakness:** This information is related to CWE and describes the category of the weakness.
    - **Description:** Any other key information about the vulnerability, like file names, network ports, names of vulnerable services, and so on.
  - **Affected Product:** Details about the affected products.
    - **Vendor:** Name of the affected product vendor.
    - **Product:** Name of the affected product itself.
    - **Version:** Affected version of the product.
    - **Module:** Particular affected part of the product.
    - **Platform:** Target operating system (possibly other type of platform) of the affected product.
    - **Architecture:** Target architecture of the affected product. This is most commonly x86 or x64.
  - **Reference:** Reference to additional information about the vulnerability.
    - **Web:** URL of a website that contains related information.
    - **CVE:** CVE reference to a CVE article that contains related information.
    - **Credits:** Typically, the name of the person or the company that has discovered the vulnerability.
    - **Miscellaneous:** Any other reference to a source of additional information.

The taxonomy also includes categories of information that can be commonly found among the article meta-data. However, the main motivation behind this experiment was that the results could be also applied to articles that do not contain

this type of meta-data, like the articles coming from mailing-lists and blogs.

Dictionary Search

It is not uncommon to find terms in various articles that refer to the same named entities. These can be names of companies, names of commonly used products, various security terms, and so on. It is therefore possible to build a dictionary of these terms and identify various entities within the text simply by their names.

Although this might seem to be a simple and straightforward approach, it also comes with several complications. First of all, if the dictionary is supposed to sufficiently cover the set of possible named entities, it typically needs to contain a lot of entries. The information needed to create such a large dictionary might not be available and creating it manually can be a very labor-intensive task. Even if the information is available, it might very likely require some additional processing, such as erasing irrelevant parts of entries. Additionally, if the dictionary grows very large, the search process might become noticeably slow and memory-intensive.

The second complication is that various named entities can be referred to using various names, abbreviations, and so on. For example *Cross-site scripting* can be also referred to as *Cross site scripting* (without the hyphen) or simply *XSS*. So the dictionary might need to contain multiple entries for the single entity. Besides, figuring out these alternative names might also not be an easy task.

Another complication is that the search does not take into account the context in which an entity exists. This context can in some cases affect the meaning of a term, making it difficult to put it into the right category.

The dictionary created for the purposes of this thesis contains the most common company names, names of operating systems, names of popular software products, and basic security and computer terms. The information was collected from the following websites:

1. http://en.wikipedia.org/wiki/List_of_computer_term_etymologies
2. http://www.sans.org/security-resources/glossary-of-terms/
3. http://download.cnet.com/windows/most-popular/3101-20_4-0.html?tag=rb_content;main
4. http://en.wikipedia.org/wiki/List_of_operating_systems
5. http://www.computerhope.com/support.htm

Collected terms were adequately shortened and mapped to the designed taxonomy in the following way:

1. Computer terms → Vulnerability Details
2. Security terms → Vulnerability Details
3. Names of popular software products → Affected Product – Product
4. Names of operating systems → Affected Product – Platform
5. Names of well-known computer companies → Affected Product – Vendor

The ontology-based gazetteer was used to search for the terms in various articles. As could be expected, only the most common terms were found this way. However, although this result might seem to be unsatisfying by itself, it could provide a base ground for application of other methods.

Knowledge-Based Rules

It is often possible to find specific common patterns in the texts that are related to the same domain. Although discovering these patterns might be more difficult than identifying the named entities, this method can eventually provide relatively good coverage, especially when appropriately combined with the dictionary search.

Probably the most significant complication is that finding the patterns requires careful reading of many texts from the target domain. Additionally, identification of many hidden domain-specific patterns might require that this task is performed by a person that is experienced and well educated in the domain of the texts. Once the patterns are identified, they need to be adequately transformed into proper rules.

Creating these rules in GATE is done by writing JAPE grammars. In order to have a sufficient base for creating the rules, it is almost necessary that the texts are preprocessed by various language analyzers. Having a dictionary of named entities available can also be very helpful. The following text describes several rules that were designed for pattern matching in articles about software vulnerabilities. The rules are based on observations from reading many of these articles [NVDA11, OVDA11, SECA11, SEFA11, SETA11, VUPA11]. The presented JAPE grammars make use of the ontology (taxonomy) described above. All the grammars share the following initial lines:

Figure 4.7: *The common header for the presented JAPE grammars.*

```
Phase: VulnerabilityIdentification
Input: Token SpaceToken Split Mention
Options: control = appelt
Template: ontology = "http://vmt.mff.cuni.cz/owlim"
Template: ontoTemplate = "http://vmt.mff.cuni.cz/owlim#${class}"
```

Identifying References

CVE numbers as well as URLs follow a specific format. The format of CVE numbers was described in Section 2.4. The following JAPE grammar can be used to recognize these formats and identify them in a text. It requires that the text has been processed by the ANNIE English tokenizer and the ANNIE sentence splitter.

Figure 4.8: *The JAPE grammar for identifying references.*

```
Rule:CVEidentification
({Token.string == "CVE-"}
 ({Token.kind == "number", Token.length == 4})
 {Token.string == "-"}
```

```
 ({Token.kind == "number", Token.length == 4})
):cve
--> :cve.Mention =
  { ontology = [ontology], class = [ontoTemplate class = "CVE"]}

Rule: URLidentification
( ( ( {Token.string =~ "http(s?)"}
      {Token.string == ":"}({Token.string == "/"})[2])
   |( {Token.string == "www"}{Token.string == "."}))
  ({!Split, !SpaceToken})+
):url
--> :url.Mention =
  { ontology = [ontology], class = [ontoTemplate class = "Web"]}
```

Identifying Details about Affected Products

Given the name of a computer company that is followed by a sequence of words that start with capital letters, it is very likely that these words form a name of a software product. Similarly, if a product name is followed by a sequence of numbers and dots, there is a good chance that this sequence represents the product version. The original named entities could be found by the dictionary search, or by other means.

The following JAPE grammar can be used to recognize the described patterns. The text needs to be again preprocessed by the ANNIE English tokenizer and the ANNIE sentence splitter.

Figure 4.9: *The JAPE grammar for identifying details about affected products.*

```
Rule:ProductIdentification
Priority: 2
{Mention.class == [ontoTemplate class = "Vendor"]}
(({Token.orth == "upperInitial"}|{Token.orth == "allCaps"})+
):product
--> :product.Mention =
  { ontology = [ontology], class = [ontoTemplate class = "Product"]}

Rule:VersionIdentification
Priority: 1
{Mention.class == [ontoTemplate class = "Product"]}
(({Token.kind == "number"}|{Token.string == "."})+
):version
--> :version.Mention =
  { ontology = [ontology], class = [ontoTemplate class = "Version"]}
```

Identifying the Type of Weakness

Another observation was that the words *vulnerability*, *vulnerabilities*, *attack*, and *attacks* often appear after a type of weakness that is related to the vulnerability

described in the text. Examples of such patterns are *SQL injection attack, cross-site scripting vulnerability,* and so on. The JAPE grammar cannot rely on the capitalization of the letters in this case. However, the sequence that determines the type of weakness is very likely a sequence of adjectives, nouns, and proper nouns. This assumption can be represented by the following JAPE grammar. It requires that the text is additionally processed by a POS tagger.

Figure 4.10: *The JAPE grammar for identifying the type of weaknesses.*

```
Rule:WeaknessIdentification
(({Token.category == "JJ"}|{Token.category == "NN"}
 |{Token.category == "NNP"})+
):weakness
( {Token.string == "vulnerability"}
 |{Token.string == "vulnerabilities"}
 |{Token.string == "attack"}|{Token.string == "attacks"})
--> :weakness.Mention =
  { ontology = [ontology],
    class = [ontoTemplate class = "TypeOfWeakness"]}
```

Machine Learning

GATE works primarily with annotations. We used annotations that covered the whole texts in order to perform the classification task. A machine learning algorithm was then used to identify the features of these annotations.

However, machine learning in GATE can be also used to identify new annotations. Given a set of properly annotated text documents, a machine learning algorithm can be trained based on the characteristics of the annotations and subsequently used to annotate other texts. These characteristics can be determined for example by the tokens that are covered by the annotations, or the tokens that are in proximity of the annotations. The goal of this experiment was to train a statistical model based on these tokens using the following characteristics:

- capitalization of letters
- part-of-speech tags
- word lemmas
- token kind – number/word/punctuation/symbol

Because there was no training set available, it was necessary to create it. Part of the annotation process could be done automatically using the dictionary and the JAPE grammars described above. However, most of the annotations needed to be added manually. The resulting training set contained twenty annotated documents.

Because of the small number of documents, it was not possible to perform reliable automated evaluation. The chosen algorithm was the Support Vector Machine, which showed the best results during the classification task. The complete configuration created for this experiment can be found in Appendix D.

The algorithm was trained on the training set and subsequently applied to twenty additional documents. Reviewing the testing set showed that even with such a

small number of training documents, the algorithm was able to produce reasonable results. Figure 4.11 shows the results on one of the testing documents.

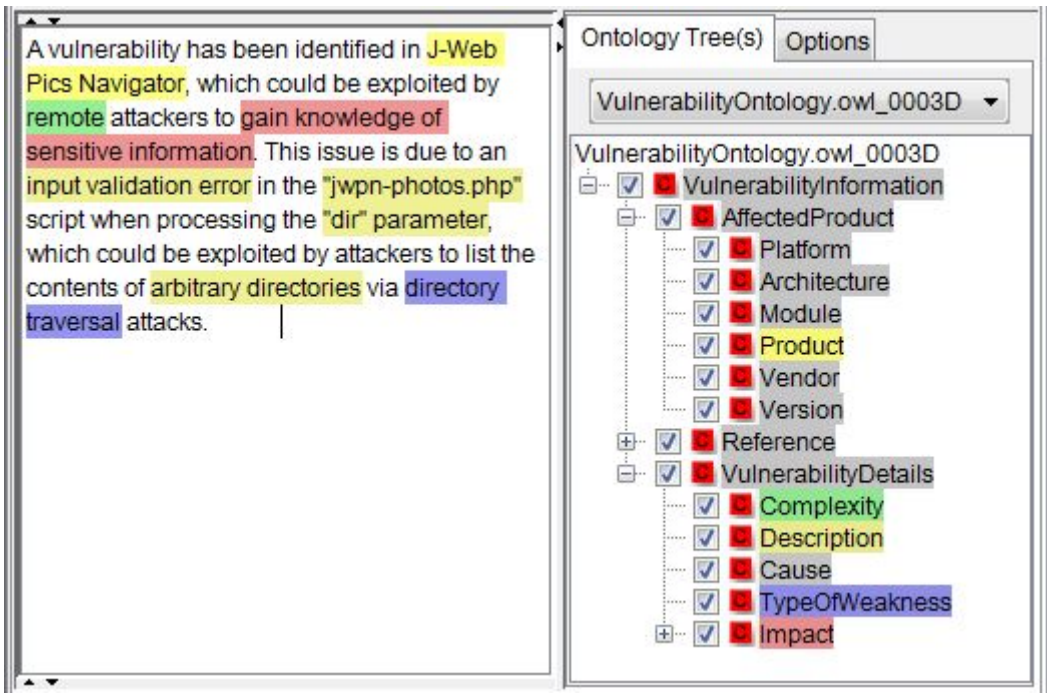


Figure 4.11: *Results of extraction of key information using machine learning presented by GATE on one of the testing documents.*

Conclusion

There are various methods which can be used to extract key information from articles about software vulnerabilities. Each of these methods has its own advantages and disadvantages. It is often possible to combine one method with another to get better results.

The first introduced method was *dictionary search*, which requires creation of a large dictionary of named entities. The second introduced method was *creation of knowledge-based rules*, which requires expert knowledge about the target domain. The last described method was based on *machine learning*, which requires a proper configuration and the presence of a training set. We could see that even a relatively small training set can be sufficient to provide reasonable results. This set can be further expanded by annotating additional documents using the trained statistical model, correcting the annotations, and adding the documents to the training set. Once the training set grows large enough, it is likely that the statistical model will be able to provide much more reliable results, and it could become eventually usable in real-world applications.

5 Information Gathering

The previous chapter has described various ways of how articles about software vulnerabilities can be analyzed. The primary information source used during this analysis was the National Vulnerability Database. The results of the analysis, however, can be also applied to articles from many other sources. There are organizations that often publish articles about certain vulnerabilities few days before these vulnerabilities are included into the NVD database. They also usually include some additional information, such as a solution description. A few of these organizations provide their articles, or some part of the information, for free.

The articles from these organizations are almost always accessible via HTTP/HTTPS protocol and the number of the articles reaches sometimes a dozen per day. The following websites were analyzed and taken into consideration as the potential sources of information for the purposes of this thesis:

1) The content is downloadable in form of XML files. There are usually several new articles each week.

- http://nvd.nist.gov/download.cfm

2) The content is in HTML form. The sources provide a decent average number of articles per day.

- http://www.securityfocus.com/vulnerabilities
- http://www.security-database.com/view-all.php?date=All&sev=All&type=All
- http://www.securitytracker.com/archives/summary/9000.html
- http://securityreason.com/security_alert/

3) The content is in HTML form. Part of the information is available only for the subscribed customers. There is about a dozen of new articles each day.

- http://secunia.com/advisories/historic/
- http://www.vupen.com/english/security-advisories/

4) The content is downloadable as a SQLite database. The information is frequently updated, but not always complete.

- http://osvdb.org/database_info

5) The content is in HTML form. The source targets only CISCO products.

- http://www.cisco.com/en/US/products/products_security_advisories_listing.html

6) The content is in HTML form. The sources list only a few of the most critical vulnerabilities

- https://www14.software.ibm.com/webapp/set2/sas/f/redAlerts/
- http://www.symantec.com/business/security_response/landing/vulnerabilities.jsp
- http://www.securitywizardry.com/radar.htm
- http://www.kb.cert.org/vuls/bypublished
- http://www.microsoft.com/technet/security/advisory/RssFeed.aspx?securityadvisory
- http://www.ca.com/us/vulnerability-management.aspx

7) Blogs containing discussions about various software vulnerabilities. The

information has almost no structure.

- http://seclists.org/fulldisclosure/
- http://blogs.technet.com/msrc/

8) Mailing lists. The information has almost no structure.

- http://www.terena.org/activities/tf-csirt/mailing-list.html
- http://seclists.org/dailydave/

5.1 Sources of Information

The main criteria when choosing the most appropriate source were the average number of articles published per day, the accessibility of the articles, and the structure level of the information. Six sources were selected after careful consideration.

The National Vulnerability Database

http://nvd.nist.gov/download.cfm

This source is commonly referred to as NVD and it was used as the primary source of information for the analysis part of this thesis. The database is maintained by the U.S. Government and is related to most of the standards described in Section 2.4. The information can be browsed on the NVD website, but can be also downloaded in form of XML files. There is a separate XML file for each year. Additionally, there is a single XML file containing recently published or updated articles. The information includes CVE identifier, date when the article was published, last date when the article was modified, short description of the vulnerability, list of affected software products, list of references to other sources, CWE category identifier, and base CVSS metrics provided by NVD.

The Open Source Vulnerability Database

http://osvdb.org/database_info

This source is often referred to as OSVDB. As the title suggests, it is an open source database created and maintained by a community of security researchers. The database contains information about a large number of vulnerabilities including very old ones. The information can be browsed on the OSVDB website, but is also downloadable in various forms. Registered users can download it in form of CSV file, SQLite database, MySQL database, or XML file. Exports into these four forms are done regularly. The SQLite database has so far proven to be the most suitable and easy to use form. The website also provides functionality called OSVDB API, which can be used to query the database using specifically crafted URLs. The results are in XML form. Unfortunately, the XML form references many parts of the information, using IDs, rather than including the information directly. This, together with the limited number of permitted requests per day, makes the API more difficult to use.

The information gathered from OSVDB includes OSVDB identifier, title, date when the vulnerability was discovered, date when the vulnerability was disclosed, date when the article was published, last date when the article was modified, short description of the vulnerability, technical description, solution description, list of affected software products, list of references to other sources,

credits, OSVDB classification of the vulnerability, and base CVSS metrics provided by OSVDB.

Secunia

http://secunia.com/advisories/historic/

Secunia is a Danish computer security company, which performs an active research in the field of software security and provides services to help users identify vulnerable software on their computers. One of its major products is called Secunia PSI, a small client application, which can be installed on a PC running Microsoft Windows. The application scans the computer for installed software, communicates with the Secunia server via the Internet, and provides up-to-date information about the security state of the computer system.

The company also publishes articles about the discovered software vulnerabilities on a daily basis. Part of the information in the articles is publicly available. This includes Secunia identifier, title, date when the article was released, last date when the article was modified, criticality level of the vulnerability, list of possible impacts, short description, solution description, list of affected software products, list of related CVE numbers, and credits.

Security Focus

http://www.securityfocus.com/vulnerabilities

Security Focus is a community based website, which provides very detailed information about the recently discovered software vulnerabilities. The list of affected software sometimes contains thousands of items and includes all the products that are affected and all their affected versions. A user can also subscribe to various mailing lists based on his interest. The only little flaw of the website is that, probably because of its popularity, the server takes sometimes several seconds to respond. The content of the articles is split among five different tabs. The first one includes Security Focus ID (also referred to as Bugtraq ID), related CVE number, date when the article was published, last date when the article was modified, list of affected software products, and credits. The second tab contains discussion about the vulnerability, which usually takes form of short description. The third tab contains a text describing the way how the vulnerability can be exploited. The fourth tab contains a solution description and the last tab contains a list of references to other sources.

Security Tracker

http://www.securitytracker.com/archives/summary/9000.html

Security Tracker is not as well-known as the other sources listed above, but still provides a decent number of articles. The information is publicly available and includes Security Tracker ID, related CVE number, date when the article was published, possibly also last date when the article was modified, list of possible impacts, short description, impact description, solution description, affected product, list of affected versions, and list of potential underlying platforms.

Vupen

http://www.vupen.com/english/security-advisories/

Vupen is a French computer security company, which publishes software security articles on a daily basis. The articles used to be freely available and were used as a source of information for the purposes of this work. Unfortunately, the company has recently stopped making its articles public and is therefore no more used as a source in the implementation part of this thesis.

5.2 Transformation into a Common Format

Each of the listed sources provides a slightly different kind of information. It might be often a good idea to transform the information into a common format, so all the sources can be treated the same way afterwards. The following common format was designed for the purposes of this thesis:

- **Information about the source:** The source name, the original ID of the article, and the original link to the article.
- **Title:** If the source does not provide any kind of title, the first hundred characters of the vulnerability description are used instead.
- **Release date:** To keep a complete history of articles and avoid unnecessary complications, each update of an article is treated individually as a new instance. The only reliable connection between the instances is the original ID, which can be later used to connect them together. The release date of the particular instance is therefore identified by the last update date of the article.
- **Risk:** The risk can take one of values *Low*, *Medium*, *High*, *Critical*. If the source provides some kind of information about the criticality level of the vulnerability, it is adequately mapped to one of these values. If no such information is available, but the source provides base CVSS metrics, then the CVSS Base Score is used to determine the value of this element. Otherwise, this element is empty.
- **Summary:** The summary combines various descriptions that are present inside the article. This does not include the solution description.
- **Solution:** If the source does not provide any solution description, this element is empty.
- **Affected products:** The affected software products are extracted as strings, which in most cases contain the name of the product vendor, the name of the product itself, and the affected version.
- **References:** There are two major types of references - web references and CVE numbers. They need to be sometimes extracted from various description texts of the article.
- **Credits:** The element usually contains information about the person that has discovered the vulnerability. If the source does not provide the information, this element is empty.
- **Base CVSS metrics:** If the source does not provide any CVSS metrics, this element is empty.

The XML schema describing the common format as well as the XSLT transformations for the individual sources can be found on the enclosed CD.

5.3 Methods of Gathering Information

The last four described sources provide the information only in pure HTML form, which needs to be preprocessed before it can be transformed into the common format. This process is often referred to as web scraping and its difficulty strongly depends on how dynamic and well-formed the input HTML is.

An important thing to notice is that a nice looking web page does not necessarily mean nicely structured or even well-formed HTML code. The appearance often depends more on the CSS styles than on the HTML behind it and there is a variety of ways how the HTML code for a page with the same appearance can be written. For example, if a paragraph *B* appears to be below a paragraph *A* on the web page, it might be situated this way in the HTML code, but it might be just as well inside the paragraph *A*. Many web browsers are even capable of interpreting broken or invalid HTML code.

Another important thing is that many web pages have dynamic content. The HTML code is usually dynamically generated by the requested server and its structure might depend on various factors. It is most commonly determined by the data that are currently being presented. This might lead to some unexpected results. For example, a presence of an HTML element might be dependent on the presence of some data inside the underlying database. The HTML might in some cases change also while the web page is being presented. This might be due to some Javascript functionality or a dynamic AJAX call.

A good first step might be to take a deeper look at the website and take time to figure out how the content looks like in various situations. This is probably the most convenient way how to find out which content is and which is not interesting. Each website usually follows its own pattern and some parts of it might be non-standard or unexpected. An article can be for example split among multiple web pages. Sometimes this step can lead to a discovery that the interesting content is also available in a more structured form, like XML. When browsing a website using Mozzila Firefox, symbol  at the right-hand site of the URL bar indicates that there is an RSS feed available. Other browsers usually indicate this information as well in their own way. The content received from an RSS feed is in XML form and might simplify the next steps.

The RSS feed may of course not be related to the relevant information, or may contain only a part of the information. If it is related, it usually contains at least the URL references to the new articles. These references might be very helpful, because they determine which web pages that are present on the website are relevant.

The references might be also listed on a dedicated web page. This is also the case of our four targeted sources. The extraction of the references is usually a simple web scraping task and consists of extraction of the relevant HTML block and retrieval of the URL references from the child "A" elements.

After downloading the web pages from the URL references, the next step is to extract the information from the gathered documents into a structured and meaningful form. There are both free and commercial tools, which can be configured to automatically perform this task. They usually transform the document into well-formed XHTML format and present it to the user in a convenient way, so he can select the interesting parts. The simple ones usually do not handle dynamic web

pages very well. The more advanced ones use various advanced techniques, including machine learning, and are usually expensive.

With some programming skill, it is also possible to handle the task without using any of those tools. Using a programming language directly gives a lot of flexibility, but might be also difficult and time consuming. The main goal when creating the following application was to provide an environment where adding a new web source could be done programmatically in a simple and convenient way.

5.4 The Advisory Updaters Application

Articles about software vulnerabilities are commonly referred to as advisories. This is mainly in case when they also contain some kind of solution description. The application is written in C# .NET and it is designed to be usable for regular and automatic downloading and processing of new advisories. The six advisory sources described above are already integrated into the application.

Adding a new advisory source typically consists of implementing two methods and one XSLT transformation. The application is also designed to be customizable, so for example programmers that are not familiar with writing XSLT can also define a specific custom method and implement the transformation using C#. There are also several helper classes that can be useful to a programmer when adding the custom code.

Generally speaking, the application allows the programmer that wants to add a new advisory source to focus on solving the task itself rather than dealing with various technical details. Before proceeding with the User Guide, we will cover the important parts of the application work-flow. It is not important for a typical user to understand all of the work-flow details, but it is useful to have the basic idea.

The Application Work-Flow

The application is configured to store the update information inside a particular folder. The path to the folder can be changed in the configuration file. The folder contains sub-folders, which contain the information from the individual runs of the application. The names of these sub-folders have a format of a time-stamp `yyyy-MM-dd_HH-mm-ss`. Each of these sub-folders contains an event log file called `eventLog.txt`, an error log file called `errorLog.txt`, and a separate source-specific sub-folder for each of the advisory sources.

When the application is started, it creates a new folder with the current time-stamp, then looks for the folder from the last run and copies a file called `lastUpdateInfo.xml` from each of the source-specific sub-folders while creating the corresponding sub-folders in the new directory. The information from this file is used to identify which advisories need to be downloaded and to avoid processing of the advisories that have already been downloaded and processed during the previous run. If no last update folder exists, the default settings specified in the configuration file are used instead.

The update for each of the sources is then started in a separate thread. Each of these threads performs five steps during its lifetime. The result of each step is passed directly to the next step as well as saved to the hard-drive under the source-specific folder. Saving to the hard-drive is mostly done for debugging and logging

purposes. The five steps are:

Links Extraction

The result of this step is a list of URLs which contains links to the documents that are potentially new and need to be further processed. The process usually involves some simple web scraping, but in some cases (NVD, OSVDB) it might also just return a single static reference, which points to a file that contains all new advisories. The result is stored in a file called `links.txt`.

Advisory Download

The default implementation of this step simply performs a download from the extracted URLs. The result is stored under a folder called `HtmlAdvisories`.

Advisory Parsing

Customization of this step is almost always necessary because it needs to be tailored to the specific source. The downloaded documents need to be processed and the information needs to be extracted into XML form. The result is stored in a file called `Advisories.xml`.

Advisory Transformation

The default implementation of this step transforms the extracted advisories using the provided XSLT transformation. The result is stored in a file called `transformedAdvisories.xml`.

Advisory Filtering

This final step is common for all sources and cannot be customized. It serves two purposes. The first is to validate that the transformed advisories are in the correct format. In order to test this, the advisories are validated against the XML Schema that describes the common advisory format. The second purpose is to eliminate the advisories that have already been processed during the previous run. The identification of the new advisories during the links extraction is usually done using the date of the latest processed advisory. The way to tell which advisories are new is usually to take the advisories that were published on the same date or later. To avoid getting the same advisories again, their SHA-1 hash codes are stored inside the file `lastUpdateInfo.xml`, so they can be compared to the hash codes of the advisories processed during the next run. The result of this step is the final result of the application run and is stored in a file called `filteredAdvisories.xml`.

User Guide

The application runs on Microsoft Windows and requires .NET 4.0. The machine that is running the application also needs to be connected to the Internet in order to make it possible for the application to connect to the advisory sources and download the data. If there is a firewall blocking the traffic, it needs to be configured to allow outbound HTTP connections to the following web sites:

- nvd.nist.gov

- osvdb.org
- secunia.com
- www.securityfocus.com
- securitytracker.com

The application can be started directly using the executable file `AdvisoryUpdaters.Application.exe`, or by running the batch file `UpdateAdvisories.bat`. The batch file starts the application as a low priority task, so the user can continue using the machine while the application runs in the background.

The application configuration is stored in a file called `AdvisoryUpdaters.Application.exe.config`. It is recommended to take a look at it before starting the application for the first time. The configuration contains the following application settings:

- **AdvisoryUpdateDirectory:** A path to the directory where the update information is stored. The directory needs to exist. The path can be absolute, or relative to the root folder of the application.
- **MaximumNumberOfAdvisoriesToProcess:** The maximum number of advisories that are processed during a single run per each source. The detection whether the advisories count exceeds this limit is most commonly done during the links extraction. If the advisories count is too high, then only the links to the oldest advisories are taken, so the limit is met.
- **Settings ending with the suffix DefaultUpdateInfo:** The default last update information for the individual sources. This is most commonly a date from which the update should start. The advisories that are older than this date are not downloaded, even if there is last update information from the previous run available.
- **Settings ending with the suffix LinkPrefix:** Changing these settings is usually not required and it is recommended only to advanced users. A situation when it might make sense is for example if the user wants to get some older advisories from NVD. He might need in that case to change the value of NVDLinkPrefix, so it points to the file that contains the older advisories.

The Technical Background

For a programmer that wants to add a new advisory source updater, it might be helpful to understand some of the important technical aspects of the application.

Execution Process Locking

When the application is started, it creates a file called `lock.txt` under the application root directory. If the file already exists, the application terminates immediately to prevent any issues that could be caused by running two instances of the application in parallel. When the application finishes, or if an error occurs during the execution, the file is deleted from the hard-drive.

Logging

The source-specific threads log the progress information about the five work-flow

steps into a single file called `eventLog.txt`. The access to this file is synchronized to prevent collisions. If an error occurs, a short description about this error is logged here as well.

Additionally, a detailed description about any error that has occurred is saved into a file called `errorLog.txt`. This description includes a detailed stack-trace of the thrown exception, which makes it easier to track down the source of the error.

The reason behind having such a logging mechanism is that many errors can occur even after the application has been regularly running for some time and might become especially useful if the occurred error is an occasional event which is difficult to reproduce.

Architecture

The architecture makes use of a concept called *dependency injection* [DEIN11]. An integration of this concept is done using a .NET library called *Castle Windsor* [WIND11]. The application consists of three main projects and an additional project for each of the advisory sources. The three main projects are:

1. **Core:** This project is a class library and contains all of the interfaces, models, and data that are common for the whole application. All of the other projects reference this project to gain access to the interfaces and models. Thanks to *dependency injection*, the other projects can use the interfaces without a need to have any knowledge about the particular implementation behind. The project also includes a static class that contains most of the constants used in the application.
2. **Services:** This project is also a class library and contains most of the common implementation. This includes the default implementation of the five work-flow steps, where each step has its own class. It also includes classes which control the overall update process and some helper classes, which will be described later.
3. **Application:** This project is a console application that contains the *Main* function, which is executed when the application is started. The execution involves process locking, process initialization, resolution of components, and spawning of the source-specific threads. The resolution of components is done using dependency injection and involves creation of various objects using appropriate interfaces and their implementations. It is therefore needed that this project references all the other projects including the source-specific ones.

Figure 5.1 shows the dependencies among the projects.

The HTML Digger Helper

This class contains several methods, which can be useful when writing the custom implementation. It is therefore helpful to be aware of their presence. The most important methods are:

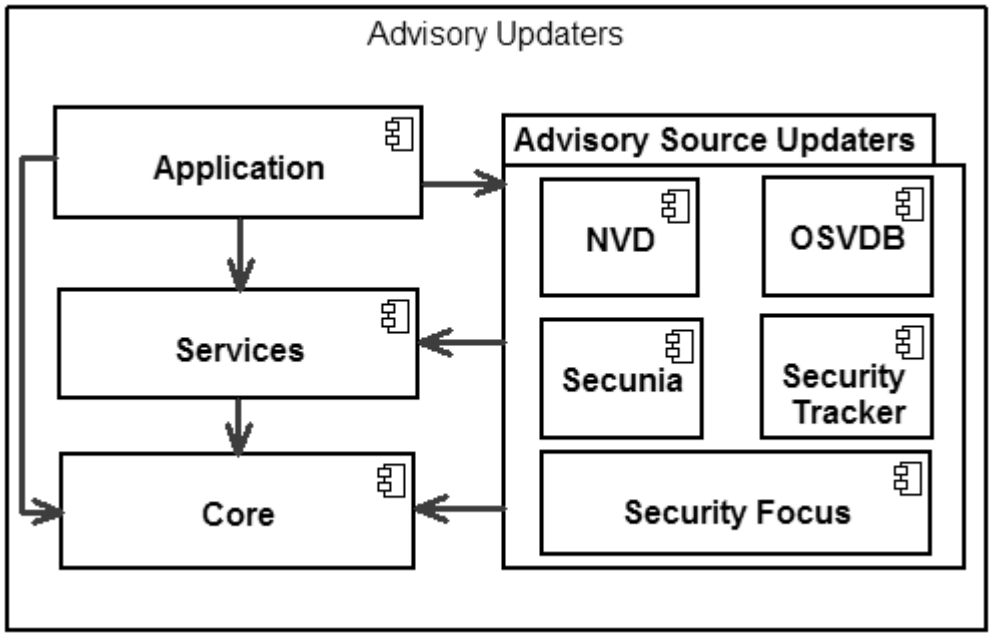


Figure 5.1: *The dependencies among the projects of the Advisory Updaters application.*

- **RunDownload:** The method downloads the HTML from the specified URL as an UTF-8 string.
- **ParseElement:** The method parses the provided HTML string and transforms it into a .NET XDocument object. The XDocument can be then queried using various ways provided by .NET, including XPath, Linq to XML, and so on.
- **ExtractElement:** The method extracts the HTML element specified by an XPath from the given HTML string. The method is suitable for quick extraction of a block that contains the relevant information. This can significantly reduce the size of the HTML string, which is then subsequently parsed. The method does not perform any parsing, but rather treats the HTML as a string. Its usage is not recommended for HTML inputs that are potentially broken or invalid.

The File Storage Helper

This class contains the methods used to interact with the hard-drive. This includes saving and retrieving the information about the last update, writing into the log files, saving the results of the work-flow steps, etc. Please review the implementations of the existing advisory source updaters to find out how to use these methods, or read the code documentation for the `IFileStorage` interface.

Validation of the Configuration File

If the application run terminates in an unexpected way, it might leave in some cases the file `lastUpdateInfo.xml` in an inconsistent state. To make the application capable of recovering from such error during the next run, the file

`lastUpdateInfo.xml` is validated against an XML Schema before the information from the file is used. This schema can be found on the enclosed CD. If the file is not valid, the file from the previous run is used instead. If that file is not valid either, the application looks at the run before the previous run, etc., until it finds a file that is valid. If no such file exists, the default configuration is used instead.

The Security Focus Downloader

The download from the Security Focus is special in two ways - the content of the articles is split among multiple pages and the server sometimes takes longer time to respond. Therefore a custom implementation for the download from this source was required. To deal with the second issue, the download runs simultaneously in multiple threads using a thread pool. This approach has proven to be useful and might be also later integrated into the default download implementation.

Adding a New Source

One of the main goals of the application is to make adding of a new source as simple as possible while keeping the flexibility that comes with the direct usage of a programming language. The application provides a fully integrated template for the whole work-flow, where a lot of functionality can be used as it is or adapted as needed. The programmer is also encouraged to use the helper classes described above when implementing the custom methods.

Adding a new source typically involves implementing a method for extraction of links to relevant documents, a method for parsing the documents into XML form, and writing an XSLT file for transformation into the common format. Although it is possible to parse the documents directly into the common format and use the default XSLT transformation, it is recommended to first extract the information into an XML that reflects the nature of the document to simplify the thinking process.

Adding a new source is done programmatically, so the programmer needs to first acquire the application source codes. The development environment that was used to create the solution was Visual Studio 2010. After opening the solution, the programmer adds a new project under the virtual folder called AdvisorySources. The name of the project should by convention have the format `AdvisoryUpdaters.{Unique_source_identifier}`, where `{Unique_source_identifier}` is a reasonably long string that uniquely identifies the new source.
The next steps are as follows:

- The programmer adds project references to the new project for the following projects: `AdvisoryUpdaters.Core`, `AdvisoryUpdaters.Services`.
- The programmer adds classes for link extraction and document parsing. Typical names for these two classes are `LinkExtractor` and `AdvisoryParser`. It is recommended that the `LinkExtractor` class is derived from class `LinkExtractorDefault` and the `AdvisoryParser` is derived from class `AdvisoryParserDefault`. The following steps require that the classes are derived from their default implementations.
- The programmer adds constructors to the new classes, which should have the following format:

```
public {Class_name}(
      IHtmlDigger htmlDigger,
      IFileStorage<UpdateInfoType> fileStorage,
      IAppConfiguration appConfiguration)
: base(htmlDigger, fileStorage, appConfiguration)
{}
```

The constructors can contain additional initialization if needed.

- The programmer overrides the method `StartExtraction()` in the `LinkExtractor` class. This method should call `_fileStorage.LogLink(Uri)` for each extracted URI link and call `_fileStorage.FlushLinks()` at the end. If the method is used to identify the last update information (like for example the date of the last update), then `_fileStorage.SaveLastUpdate(update_information)` should be called to save this information.
- The programmer overrides the method `Parse(IEnumerable<Tuple<FileInfo, Uri>> files)` in the `AdvisoryParser` class. The method gets a list of downloaded documents together with their original URI addresses as a parameter. This method should call `_fileStorage.SaveOriginalXMLAdvisory(XElement)` for each parsed document and call `_fileStorage.FlushOriginalXMLAdvisories()` at the end.
- If the documents are not parsed directly into the common format, the programmer needs to create a folder called `Data` inside of the new project and add his XSLT transformation under this folder. He also needs to set the file to be copied into the Output Directory.
- The programmer can add additional classes and override additional methods, if that is required. It is always recommended to make the classes derive from their default implementations. For additional details, please review the implementation of the already added advisory source updaters, or read the code documentation of the following interfaces: `ILinkExtractor`, `IAdvisoryDownloader`, `IAdvisoryParser`, `IAdvisoryTransformation`.
- The programmer adds a class under the new project that is derived from `AdvisoryUpdaterDefault`. The programmer also needs to add a constructor to the class, which has the following format:

```
public {ClassName}(
      IFileStorage<UpdateInfoType> fileStorage,
      LinkExtractor linkExtractor,
      IAdvisoryDownloader advisoryDownloader,
      AdvisoryParser advisoryParser,
      IAdvisoryTransformation advisoryTransformation)
: base(fileStorage, linkExtractor, advisoryDownloader,
       advisoryParser, advisoryTransformation)
{
```

```
        XSLT_FILE = "NewSourceTransformation.xslt";
    }
```

The XSLT_FILE needs to be specified if the programmer provided an XSLT transformation. If the programmer has added any additional classes besides the `LinkExtractor` and the `AdvisoryParser`, he needs to replace the interface names by the class names accordingly.

- As the last step, the programmer needs to enable the new source updater. He does so simply by adding a project reference for his new project to the `AdvisoryUpdaters.Application`. The new source is then automatically included.

Additional notes:

- To gain access to the functionality of the HTML Digger helper, the programmer can make use of the object called `_htmlDigger` from any part of his code.
- To gain access to the application configuration interface, the programmer can make use of the object called `_appConfiguration`, in which case he should add the respective configuration entries into the configuration file.
- The programmer can use the method `_fileStorage.LogError(UpdaterException)` to log errors (for example in a try-catch block)
- The programmer should call the method `_fileStorage.SaveLastUpdate(update_information)` at least once inside of his code.

Parsing

Although most of the steps described above are rather simple and straightforward, there is one task, which is highly non-trivial. It is the parsing itself. If the information does not come directly in a form of XML, it needs to be somehow extracted from the source documents. The documents are most commonly HTML pages. The programmer can use the HTML Digger helper and make use of many powerful features that come with Linq to XML in that case. He can also use other tools like Html Agility Pack or Fizzler for querying the documents. Firebug plugin for Mozilla Firefox can be very helpful when analyzing the target HTML structure.

It is usually a good idea to use XPath queries to extract the information. However, the structure of the target page often varies based on the data that are being presented. Because of the dynamic content, it might not be possible to perform the querying directly. To deal with this issue, it is often possible to identify unique invariable patterns within the HTML, like header texts or CSS class names. If used carefully, these patterns can be very helpful for localizing particular information, which can then be extracted using the XPath.

But getting the information is not the end of the process. Various data, like for example dates, can be presented in various formats. It is recommended to unify these formats for further processing. Regular expressions can be useful in many cases for

data validation. They can help catching errors early enough to determine their cause.

It is always recommended to log any potential errors. Many errors can have rare causes and can occur even after the application has been successfully running for several days. Until most of these errors are fixed, it might be necessary to regularly review the error logs.

Debugging

It is very rare that a code would work as it was intended right after it had been written. Some additional effort is usually required to find and eliminate bugs. This process usually involves running the application at least several times and inspecting the execution flow. However, in case of the Advisory Updaters application, this would normally mean going through the whole work-flow every time, including more time-consuming steps such as the advisory download. To avoid this sort of delays, the programmer can change the application behavior and make it work with the data from the last update. This can be done in two simple steps:

- The programmer needs to review the Main function, comment out the line
  `_thisRunDirectory = myUpdateRunner.InitializeRun();`
  and uncomment the line
  `_thisRunDirectory = myUpdateRunner.ContinueLastRun();`
- The programmer needs to review the `UpdateAdvisories` method of the class `AdvisorySourceUpdater`, which is situated under the project `AdvisoryUpdaters.Services`. The programmer then needs to comment out any undesired work-flow steps. This usually involves commenting out commands `updater.StartExtraction();` and `updater.Download();`

These two steps will cause the updater to use the information stored inside the last update folder instead of downloading new data every time.

Troubleshooting

The application terminates right after it has been started

In some cases, when the application is terminated in a forced way, the file `lock.txt` might remain present even if the process is not running. It is therefore recommended to check the existence of the file in case when the application does not even pass the starting phase.

6 Vulnerability Management Tool

Chapter 4 has described various ways how the articles about software vulnerabilities can be analyzed, how the information from them can be extracted, and how this all can be used in order to allow user to search for the information he is interested in. The goal of the Vulnerability Management Tool project was to integrate some of these ideas and provide a functional and user-friendly application that could be used for importing, analyzing, and filtering of the advisories collected by the Advisory Updaters tool described in Chapter 5.

Java was chosen as the programming language to enable a simple integration of Lucene and GATE into the project. The application therefore requires JRE 1.6 to be installed on the target machine. All the other dependencies are included in the application and do not require any installation. The application can be simply copied to the hard-drive. However, it is very important that the application is started from a writable medium to make it possible to import new advisories, perform the analysis, and so on.

The distribution of the application also includes the Advisory Updaters tool. By default, the tool is configured to store the advisory updates inside the same directory which the application imports the new advisories from. It is possible to simply start the Advisory Updaters tool, wait until it finishes the update, start the Vulnerability Management Tool application, and click on the import button. The new advisories will be automatically identified and imported into the application. More information about the usage of the Advisory Updaters tool can be found in Section 5.4. The following chapter will guide you through the functionality of the Vulnerability Management Tool application.

6.1 User Guide

This guide focuses on the distribution of the VMT application that can be found on the enclosed CD. The main purpose of this distribution is to demonstrate the concepts described in this work.

Both the VMT application and the Advisory Updaters tool have been developed on Windows 7 32-bit and tested also on Windows 7 64-bit. Although the VMT application should run on any system with JRE 1.6 installed, the compatibility cannot be guaranteed. The Advisory Updaters tool requires .NET 4.0 and it is recommended to be run on Windows 7.

The distribution comes with several advisories already imported and several additional advisories downloaded via the Advisory Updaters tool. It is possible to import these additional advisories when starting the VMT application for the first time. If the underlying operating system is compatible with the Advisory Updaters application and the Internet connection is available, it is also possible to download new advisories by executing the file `UpdateAdvisories.bat`. These advisories can be then subsequently imported into the VMT application after the download finishes.

The VMT application can be started via `VMT.jar`. Please start the application from the directory where it is situated to ensure that all paths are recognized correctly. This can be done either by double-clicking the file `VMT.jar` or navigating to the application folder via command line and typing `java -jar VMT.jar`. Please

note that it might take several seconds until the GUI window shows up due to the number of libraries that the application needs to load. Part of the GUI window is shown in Figure 6.1.

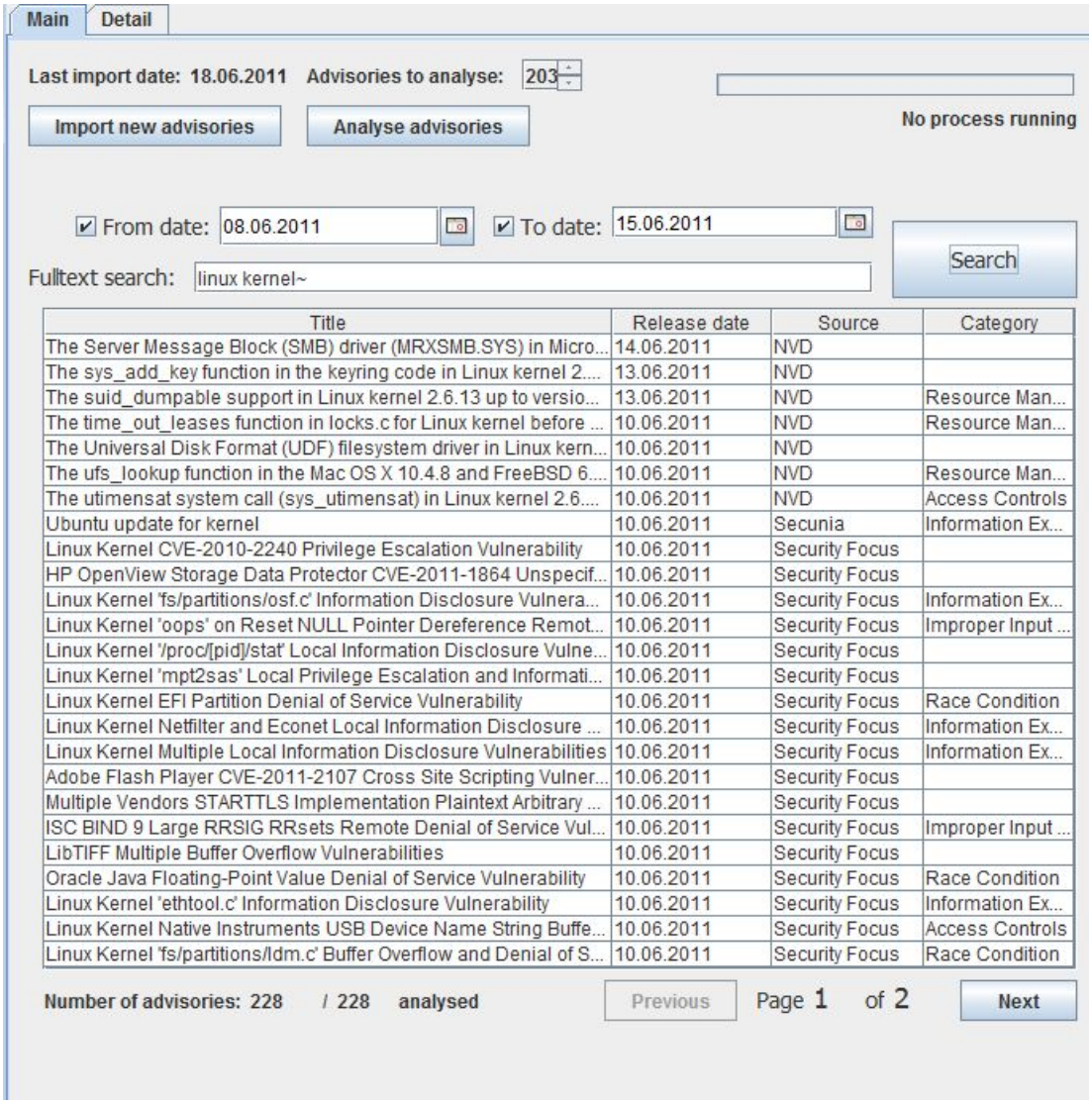


Figure 6.1: *The left-hand side of the Main tab of the VMT application.*

When starting the application for the first time, the user should draw his attention to the top left part of the application window. By clicking the button *Import new advisories,* he can import new advisories which have been previously downloaded by the Advisory Updaters tool. In order to identify which advisories are new, the application stores the full time-stamp of the last advisory update inside of its configuration file. The date of the last advisory update is presented above the button.

During the import, the user can watch the progress on the progress-bar which is also situated on the top of the window. The user can continue using the application during this process, but it is recommended to wait until the import finishes. The label under the progress-bar shows additional information about the current state of the process. The label displays the text *Advisory import successful!* after the application has successfully imported all the new advisories.

The user can now click on the button *Search* which is situated below the progress-bar. The table under the button presents the latest 25 imported advisories. The advisories are by default sorted by their release date. The user can further browse the advisories using the buttons *Next* and *Previous* situated under the table. The user can additionally choose various search criteria which will be described later in this section. One part of the filtering mechanisms that allow him to do so can be found to the left of the *Search* button.

The table displays the *title*, the *release date*, the *source name*, and the *CWE category* of the presented advisories. Advisories that have just been imported have the *category* field empty. In order to identify their categories, it is necessary to analyze them. This can be done by clicking on the *Analyze advisories* button, which is situated in the top left corner of the application window, to the right of the *Import new advisories* button. The *spin-box* above the button can be used to set the number of advisories to be analyzed. An analysis of a single advisory can take a few seconds, so it might be sometimes wiser not to analyze all the imported advisories at once. The subsequent process selects the specified number of advisories from the database and performs their classification and extraction of the key information. The selection of the advisories is not ordered by any field, so it might appear to be random. During the analysis, the user can watch the progress on the progress-bar. When the analysis finishes, the user can click on the *Search* button again to see the results. If the advisories he is interested in are still not analyzed, he can click on the *Analyze advisories* button again to analyze another group of advisories. Sometimes the category field of an advisory remains empty although the advisory has already been analyzed. This means that the analysis process was not able to identify the advisory category with a sufficient probability. The numbers in the bottom left corner of the application window display the total number of the imported advisories and the number of advisories that have already been analyzed.

When the user finds the advisory he is interested in, he can select it in the table in order to display its details on the right side of the application window. Figure 6.2 shows an example of a possible result.

The panel in the bottom right corner of the application window displays the details preview of the selected advisory. The label at the top of the panel shows the advisory *title*. The labels bellow show *the name of the advisory source*, *the advisory CWE category* (if it was identified), *the release date of the advisory*, and *the risk level of the vulnerability described in the advisory*. The *risk level* is displayed in color based on its value.

The text area below called *Summary* contains *the description text of the advisory*. If the length of the text exceeds the size of the area, the user can scroll the content using the scroll bar situated on the right side. The user can also click the button *Details* above the area to display a pop-up window which presents the description text in a more readable form. Additionally, if the advisory has been previously analyzed, the key information contained in the text is highlighted using various colors. These colors are related to the ontology described in Section 4.5 and will be described later in this chapter. An example of the pop-up window is showed in Figure 6.3.

Bellow the description text is a text area called *Solution* that contains the text describing how users can protect themselves against the attacks on the vulnerability described in the advisory. This text area can be often empty if the advisory describes

only the vulnerability itself.

The label underneath this area called *Credits* typically shows the name of the person or the organization that has discovered the vulnerability.

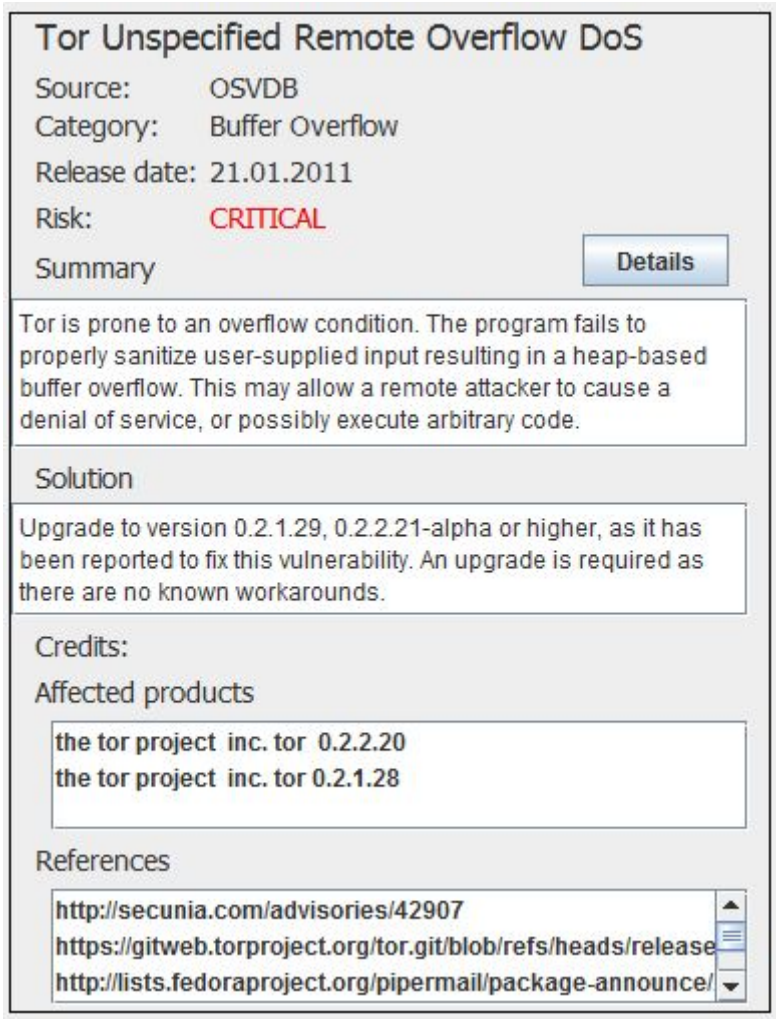


Figure 6.2: *The advisory details preview displayed on the right side of the Main tab of the VMT application.*

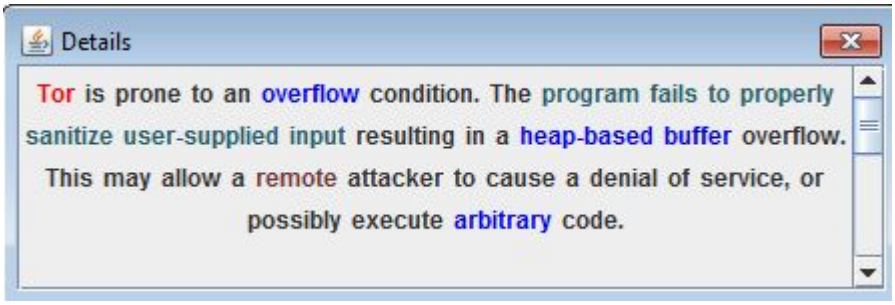


Figure 6.3: *The pop-up window displaying the description text of the advisory.*

Finally, the two list boxes situated at the bottom of the panel show *the*

software products affected by the described vulnerability and *the references to related sources of information.* Similar to the text areas, the list boxes can be vertically scrolled if the number of list items exceeds the size of the reserved area.

However, despite the possibility to scroll the content and display the *pop-up window*, the details preview might sometimes become uncomfortable to read. The user can in that case switch to the *Detail* tab in the top left corner of the application window. The information contained in this tab is the same as the information in the details preview, but it is presented in a much more readable form. The *description text* is shown directly in its colored form and the user can also see the meaning of the individual colors on the right side of the window.

Figure 6.4 shows the advisory from Figure 6.2 presented in the *Detail* tab.

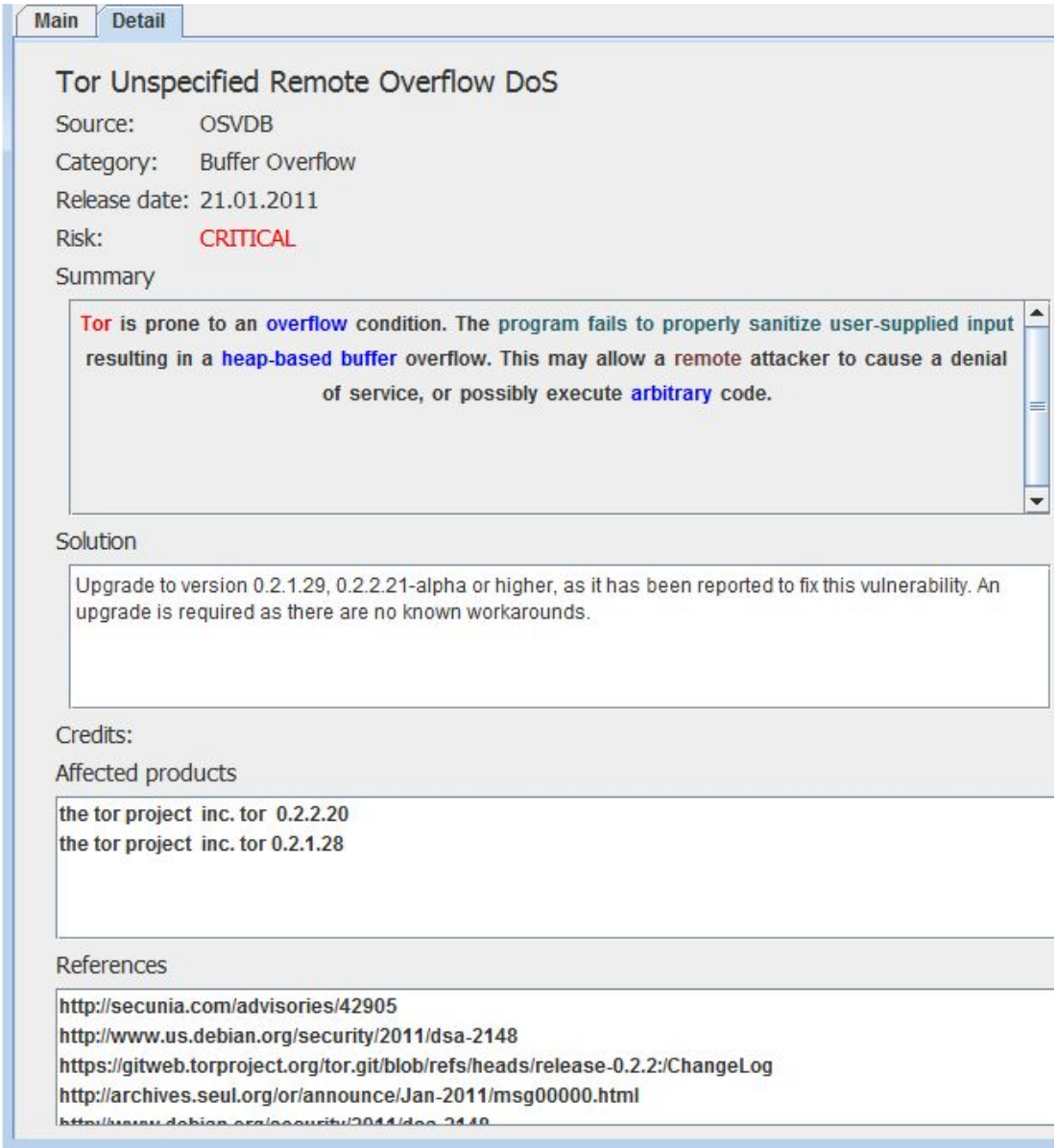


Figure 6.4: *The left-hand side of the Detail tab of the VMT application.*

Filtering Mechanisms

The main goal of the Vulnerability Management Tool application is to allow the user to filter the imported advisories and quickly retrieve the information he is interested in. In order to satisfy this requirement, the user is provided with several filtering mechanisms that he can use to specify his search criteria. Most of them can be combined together in order to search for advisories that satisfy multiple criteria. The related GUI components are usually displayed with a check box that becomes selected when the user chooses a value for the particular component. The user can then simply deselect the check box to disable the related filtering mechanism.

Figure 6.5 shows one part of the filtering mechanisms that is situated to the left of the *Search* button.

Figure 6.5: *The first part of the VMT filtering mechanisms.*

The component *From date* and the component *To date* can be used to set restrictions on the *release date* of the result advisories. The user can write the date directly into the input field in format *dd.mm.yyyy* or click on the icon on the right side of the input field and choose the date from the displayed calendar.

The component *Full-text search* provides means to search the advisories using various full-text methods provided by Lucene. The user can simply type in the keywords, and the underlying engine then searches for these words in the indexed text fields of the stored advisories. The indexed text fields are *the advisory title field*, *the advisory summary field*, *the advisory solution field*, and *the advisory credits field.* If the user separates the words by spaces, the engine retrieves results that contain any of the specified words. The user can use the word *AND* to indicate a conjunction or the word *OR* to write an explicit disjunction. The user can also use brackets '(' and ')' to write more complicated queries. It is also possible to write a name of a text field with a colon to restrict the search to the specific field. The following example shows a query that can be used to search for advisories that contain the words *buffer* and *overflow* in the title field, the word *dangerous* in the summary field, and the word *attack* in any of the indexed text fields:

```
title:buffer AND title:overflow AND summary:dangerous AND attack
```

The user can additionally search for phrases using quotes. It is also possible to search for a word in a fuzzy way by appending the character '~' to the word. The user can also use the wildcard symbols '?' and '*' to mask particular parts of the words. For additional details about the query syntax, please refer to [LQPS11].

The following query shows a more advanced example of the syntax usage:

Figure 6.6: *An example full-text query that can be passed to the VMT application.*

```
(title:database AND oracle) OR (buf?er AND overflow~ AND
summary:vulnerabilit*) OR credits:"Jeremy Brown" OR
credits:m*security
```

It might of course happen that the syntax of the query provided by the user is incorrect. The user is in that case notified by a red label that shows a short description of the error below the progress-bar.

If this filtering mechanism is used only by itself, the results are sorted by the calculated Lucene score with the best match at the top.

Figure 6.7 shows the second part of the filtering mechanisms which is situated in the top right corner of the application window.

Figure 6.7: *The second part of the VMT filtering mechanisms*

The component *Source* allows the user to search for advisories from a particular source. The list of sources is automatically updated, so if the user imports advisories that come from a new source, the source becomes automatically included in the list.

The component *Category* can be used to filter the advisories by their *CWE category*. The list contains a category only if there is at least one advisory in the database that belongs to this category.

The component *Product* allows the user to search the advisories by an affected product name. The underlying filtering mechanism implements some of the ideas related to affected products that were described in Section 4.3. The user can type in a product name, a vendor name, a product version, possibly other key strings. The engine then searches for these strings among the affected products stored in the database using the fuzzy search and the prefix search. The advisories that contain affected products which sufficiently match the input strings are then retrieved. There is also a prototype functionality which the user can enable by selecting the option *Advanced search by product.* This functionality can be used only by itself and therefore the other filtering mechanisms become automatically disabled when the user selects this option. The main feature of this functionality is that the result advisories are sorted by a score that is derived from the precision with which their affected products match the input query. Another feature is that the engine tries to recognize various short words as abbreviations and use their letters for the prefix search. This can cause, however, that many retrieved results are not truly related to what the user is interested in.

The component *Reference* can be used to search by references to other sources of information. The underlying filtering mechanism is based on the ideas related to references that were described in Section 4.3. The engine searches for the user's input string among the references stored in the database using the wildcard search and retrieves the advisories that contain the matched references.

6.2 The Technical Background

Architecture

The architecture of the Vulnerability Management Tool is similar to the architecture of the Advisory Updaters application. It also makes use of dependency injection, which is in Java provided by the Spring framework [SPRI11]. However, there are also two additional layers:

- **Data layer** provides access to the database and Lucene. All the insertion, update, and retrieval functionality is provided by this layer. The layer is implemented mostly using Hibernate, a framework for mapping the domain objects to the relational database [HIBE11]. Hibernate provides a convenient abstraction which allows the programmer to treat the data stored in the database in a similar way as he would treat standard Java objects. Additionally, Hibernate comes with an extension called Hibernate Search [HIDT11], which allows an integration of Hibernate with the Lucene library, giving it access to all the powerful full-text functionality that is provided by Lucene.
- **Analysis layer** provides the interaction with GATE and performs the classification and the information extraction. All the necessary GATE libraries are distributed with the application, so the target machine does no need to have GATE installed.

Figure 6.8 shows the dependencies among the layers.

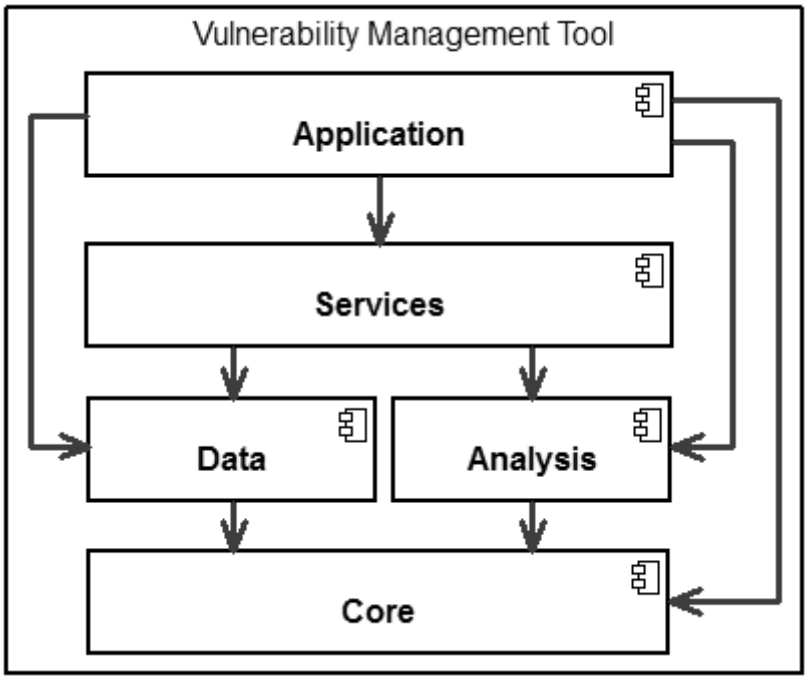


Figure 6.8: *The dependencies among the projects of the Vulnerability Management Tool application.*

Choosing the Data Storage

A suitable database engine is often the backbone of a data oriented application. Various types of database engines can serve various tasks and have various pros and cons. Because the application was going to be used for fast searching, the engine needed to have an indexing support. This requirement was satisfied by Lucene. However, it was also necessary to store various relations between the data, which led to further exploration. After a consideration of object oriented databases, such as MongoDB [MONG11], and various XML oriented databases, a decision was made to stick with a traditional relational database. Relational databases provide a very mature technology that has developed over many years and are among the most widely used database engines in the World. The final decision was to use HSQLDB [HSQL11], which is a portable SQL database engine. Thanks to this choice, the application can be simply copied to the target computer and does not require any kind of installation.

Data Import and Analysis

Before implementing the data import and the data analysis, it was necessary to design a database schema that would allow the data to be stored in the database in a coherent and structured form. The schema is shown in Figure 6.8.

Import

The first step of the application usage is typically the data import. The information gathered by the Advisory Updaters tool needs to be stored inside the database to allow efficient searching and filtering. This is a simple task in case of simple advisory elements like the title or the release date. However, elements that are parts of lists, like references and affected products, need to be stored as separate objects. Because each advisory can have multiple references and affected products, and each reference or affected product can belong to multiple advisories, these relationships are of type M:N. Each imported advisory is represented by a new instance. However, this does not count for the references and affected products. Whenever new references or affected products are to be inserted, it needs to be checked whether any of them already exists in the database. Those that are already present in the database are retrieved and accordingly mapped to the advisories. The others are first inserted into the database, and then mapped to the advisories as persistent objects.

To speed up the whole process, the references and affected products are first extracted from all the input advisories, while the mappings between advisories and their related elements are stored using hash tables. The elements can be then tested for their existence using a single "`SELECT .. WHERE .. IN (..`" query, rather than querying the database for each single element. The elements that are not among the results of the query are afterward inserted into the database. The hash tables are then used to map the elements in their persistent state back to the advisories.

Analysis

The analysis consists of classification, which assigns advisories to CWE categories, and information extraction, which assigns annotations to advisories. The functionality is primarily implemented using the GATE Embedded and makes use of the gazetteers, JAPE grammars, and machine learning models that were created as

the result of the analysis described in Chapter 4.

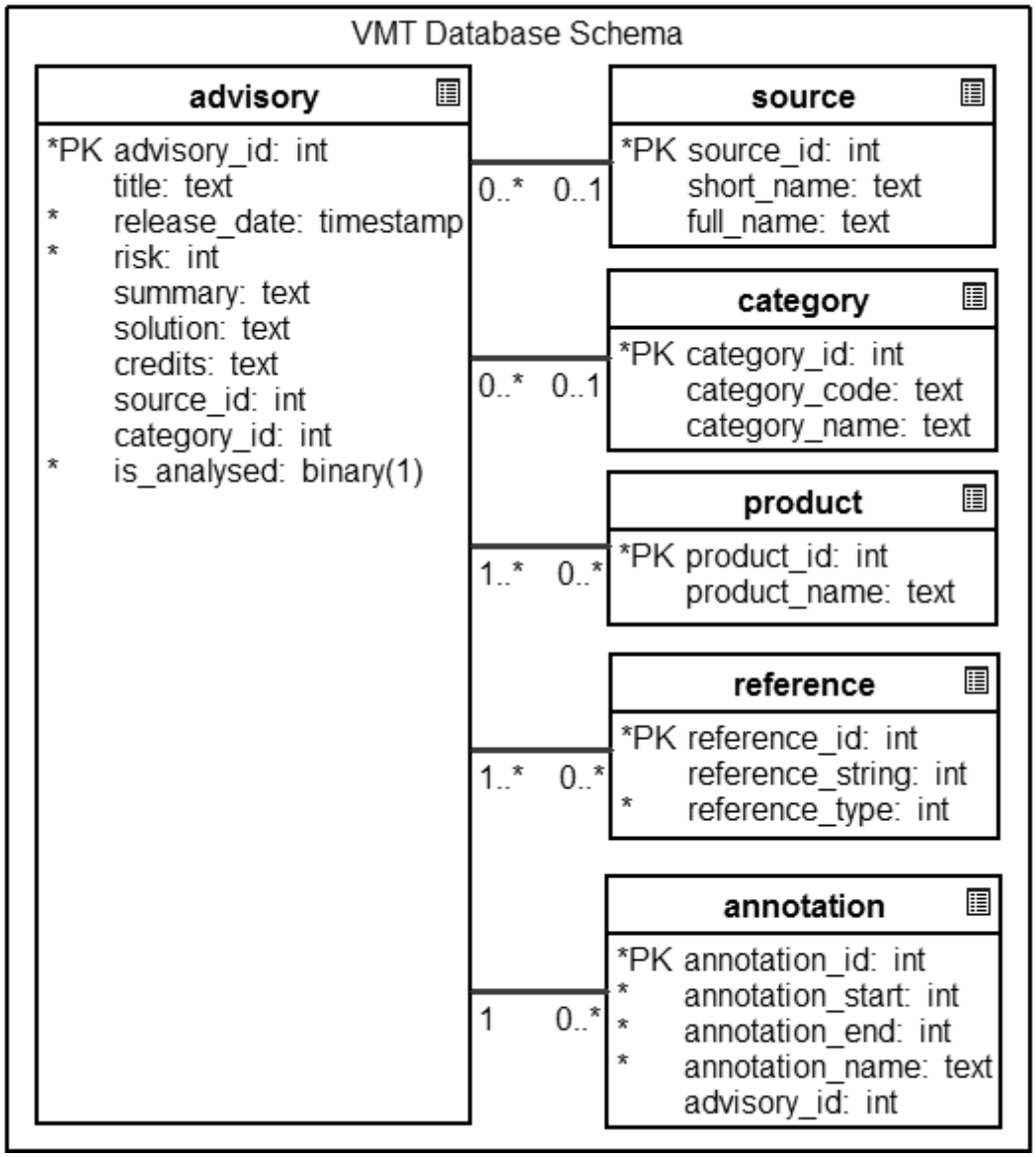


Figure 6.8: *The database schema of the VMT application. The schema does not display the tables required to maintain the M:N relationships.*

The algorithm used for the classification task is the Support Vector Machine with a linear kernel, because it showed the best precision during the experiments. The information extraction is performed using only the machine learning approach to allow better demonstration of the underlying statistical model.

6.3 Troubleshooting

The application can not be started or there are no advisories displayed

The most likely reason behind this issue is that the connection to the underlying data storage had not been correctly closed before the application was terminated. This

typically happens when the application is terminated in a forced way, for example via the task manager. To solve this issue, please delete the file `vmt.db.lck` under the directory `db/data` and start the application again.

7 Conclusion

7.1 Summary

Even when good programming practices are followed during the development of a software product, it happens many times for various reasons that the product contains a vulnerable spot after its release. Such a vulnerability can often represent a significant security threat if it is discovered and exploited by malicious subjects. It is therefore important that these vulnerabilities are identified in time and correctly reported to responsible persons. Despite the initiative to unify the format of these security reports, every source of reports publishes its articles in its own unique format. Additionally, a lot of information is usually provided in a form of unstructured text, which is not directly suitable for automated processing.

We have studied various security standards and analyzed various sources of software security reports. Using the acquired knowledge, we have implemented a tool that can be used to collect reports from various sources into a unified and structured form. We have then studied several text mining techniques which could help analyzing the reports and provide means for their filtering. During the subsequent analysis we have carried out the following tasks:

- The reports were analyzed by human observation with the focus on the contained meta-data.
- The GATE text mining framework was used to apply some of the studied text mining techniques in order to classify the reports into CWE categories. Several methods were evaluated and compared during this task. The results were described in the last part of Section 4.4.
- Several methods were applied in the endeavor of extracting the key information from the description texts of the reports. This task was also performed with the help of the GATE framework. The results were described in the last part Section 4.5.

Finally, we have implemented the Vulnerability Management Tool application that integrates various results of our analysis. The application provides a graphical interface that can be used by an average-experienced user to import, view, analyze, and filter the collected reports.

The main purpose of the presented thesis is to provide a proof of concept that certain automated analytical methods can be successfully applied to articles about software vulnerabilities in order to reduce the amount of work required for their manual processing. In the following section we list several possible future directions.

7.2 Future Work

Because a significant part of the analysis has been performed using the description texts of the reports, it is very likely that the results could be also applied to completely unstructured reports coming from various blogs and mailing-lists. When the reliability of the key information extraction reaches a sufficient level, it could be

used to extract missing meta-data, or provide additional means of filtering. By adding various relationships to the ontology designed for the purposes of this thesis, it should be possible to extract even more information from the texts. The key information could be then used, for example, for generation of summaries.

In addition to the implemented filtering mechanisms, the user of the Vulnerability Management Tool application could be also provided with an option to search for the documents that are related to a document of his choice. In that case, a possible approach could be to use document clustering described in Section 3.4 and then retrieve the whole cluster of similar documents.

Last but not least, it would be also possible to incorporate the user into the process of improving the performance of the application. A sufficiently qualified user could occasionally correct potentially incorrect results of the machine learning processes, improving thus the underlying statistical models in an interactive fashion.

8 Bibliography

[APIS99] Douglas E. Appelt, David J. Israel: *Introduction to Information Extraction Technology*, A Tutorial prepared for IJCAI-99, 1999

[BENE07] Antonin Benes: *Information Security I, II*, the lectures and the study materials, Charles University in Prague, 2006/2007

[CRSH00] Cristianini N., Shawe-Taylor J., *An introduction to support vector machines and other kernel-based learning methods*, Cambridge University Press, 2000.

[CUNN04] Hamish Cunningham: *Information Extraction, Automatic*, The University of Sheffield, 2004

[CUNN11] Hammish Cunningham: *Developing Language Processing Components with GATE Version 6 (a User Guide),* The University of Sheffield, 2011

[CVSS11] A Complete Guide to the Common Vulnerability Scoring System Version 2.0. http://www.first.org/cvss/cvss-guide.html

[DEIN11] Wikipedia. Dependency injection. http://en.wikipedia.org/wiki/Dependency_injection

[FESA07] Ronen Feldman, James Sanger: *The Text Mining Handbook*, Cambridge University Press, 2007

[FEWE08] Stefan Fenz, Andreas Ekelhart, Edgar Weippl: *Semantic Potential of existing Security Advisory Standards*, Proceedings of the FIRST2008 Conference, 2008

[GATE11] The University of Sheffield. GATE. http://gate.ac.uk/

[HAKA06] Jiawei Han, Micheline Kamber: *Data Mining: Concepts and Techniques*, Elsevier Inc., 2006

[HENL05] Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness, Michael Lester: *The Hackers Handbook*, Grada Publishing, a.s. 2005

[HIBE11] Hibernate. http://www.hibernate.org/

[HIDT11] Hibernate Development Team: *Apache Lucene Integration, Reference Guide 3.4.0 Final*, 2011

[HOTF11] Dusan Domany, Stepan Henek, Peter Kmet, Jan Stanek, Martin Zember**.** HotFuzz Software Project. Faculty of Mathematics and Physics, Charles University, Prague. 2009 - 2010

[HSQL11] HyperSQL. http://hsqldb.org/

[KAIS10] Katharina Kaiser: *Information Extraction*, the lecture and the study materials, Technical University of Vienna, 2010

[KROH11] Petr Kroha: *Text mining*, the study materials to the lecture, Charles University in Prague, 2011

[LACR96] Larkey L., Croft W.: *Combining classifiers in text categorization.* In *Proc. 19th Annual Intl. ACM SIGIR Conf. on R&D in Information Retrieval*. 1996

[LQPS11] Lucene query syntax.
http://lucene.apache.org/java/3_3_0/queryparsersyntax.html

[LUCE11] Apache Lucene. http://lucene.apache.org

[LZHT02] Yaoyong Li, Hugo Zaragoza, Ralf Herbrich, John Shawe-Taylor, Jaz Kandola: *The Perceptron Algorithm with Uneven Margins*, Proceedings of the International Conference of Machine Learning, January 2002

[MAHO11] Apache Mahout. http://mahout.apache.org/

[META11] Metasploit. http://www.metasploit.com/

[MONG11] MongoDB. http://www.mongodb.org/

[MRAZ09] Frantisek Mraz, Iveta Mrazova: *Data Mining*, the lecture and the study materials, Charles University in Prague, 2009

[NAVI09] R. Navigli: *Word sense disambiguation*: A survey. ACM Comput. Surv. 41, 2 (Feb. 2009), 1-69. DOI= http://doi.acm.org/10.1145/1459352.1459355

[NVDA11] NVD advisories. http://nvd.nist.gov/download.cfm

[OVAL11] OVAL Definition Tutorial.
http://oval.mitre.org/language/about/definition.html

[OVDA11] OSVDB advisories. http://osvdb.org/database_info

[OWL11] Wikipedia. Web Ontology Language.
http://en.wikipedia.org/wiki/Web_Ontology_Language

[PAFU10] Jan Paralic, Karol Furdik: *Text mining*, Technical University in Kosice, 2010

[PEAC11] Michael Eddington. Peach. http://peachfuzzer.com/

[PYLE99] Dorian Pyle: *Data Preparation for Data Mining*, Academic Press, 1999

[QWJS09] Stephen Quinn, David Waltermine, Christopher Johnson, Karen Scarfone, John Banghart: *The Technical Specification for the Security Content Automation Protocol (SCAP)*, National Institute of Standards and Technology, 2009

[SARA08] Sunita Sarawagi: *Information Extraction*, Indian Institute of Technology, 2008

[SECA11] Secunia advisories. http://secunia.com/advisories/historic/

[SEFA11] Security Focus advisories. http://www.securityfocus.com/vulnerabilities

[SETA11] Security Tracker advisories.
http://www.securitytracker.com/archives/summary/9000.html

[SPRI11] Spring framework. http://www.springsource.org

[SQLM11] Bernardo Damele A.G. SQL Map. http://sqlmap.sourceforge.net/

[STAL08] William Stallings, Lawrie Brown: *Computer Security - Principles and Practice*, Pearson Education Inc., 2008

[TAGS11] Automatic Mapping Among Lexico-Grammatical Annotation Models. http://www.comp.leeds.ac.uk/ccalas/tagsets/upenn.html

[VALG11] Valgrind. http://valgrind.org/

[VUPA11] Vupen advisories. http://www.vupen.com/english/security-advisories/

[WANG09] Ju An Wang, Minzhe Guo: *Security Data Mining in an Ontology for Vulnerability Management*, International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing, 2009

[WEIP10] Edgar Weippl: *Security, Internet Security, Advanced Internet Security*, the lectures and the study materials, Technical University of Vienna, 2009/2010

[WEKA11] Univeristy Of Waikato. Weka. http://www.cs.waikato.ac.nz/ml/weka/

[WILD10] Wikipedia. Levenshtein Distance. http://en.wikipedia.org/wiki/Levenshtein_distance

[WIND11] Castle Windsor. http://www.castleproject.org/container/

[YANG01] Yang Y.: *A Study on Thresholding Strategies for Text Categorization.* In *Proc. 24th Annual Intl. ACM SIGIR Conf. on R&D in Information Retrieval*. 2001

9 List of Tables

10 List of Figures

11 List of Abbreviations

CERT	.....	Computer Emergency Response Team
CERT/CC	.....	CERT Coordination Center
NIST	.....	National Institute of Standards and Technology
SCAP	.....	Security Content Automation Protocol
CVE	.....	Common Vulnerabilities and Exposures
CPE	.....	Common Platform Enumeration
CWE	.....	Common Weakness Enumeration
CAPEC	.....	Common Attack Pattern Enumeration and Classification
CVSS	.....	Common Vulnerability Scoring System
OVAL	.....	Open Vulnerability and Assessment Language
POS	.....	Part Of Speech
KNN	.....	K-Nearest Neighbor
SVM	.....	Support Vector Machine
HMM	.....	Hidden Markov Model
GATE	.....	General Architecture for Text Engineering
ANNIE	.....	A Nearly New Information Extraction System
JAPE	.....	Java Annotation Patterns Engine
NVD	.....	National Vulnerability Database
OSVDB	.....	Open Source Vulnerability Database
VMT	.....	Vulnerability Management Tool

12 Appendix A: CWE Categories

Appendix A describes the CWE categories identified in the advisories from NVD during the analysis described in Section 4.4. The descriptions are taken from the CWE database.

CWE-119: *Buffer Overflow*

The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.

CWE-134: *Uncontrolled Format String*

The software uses externally-controlled format strings in printf-style functions, which can lead to buffer overflows or data representation problems.

CWE-16: *Configuration*

Weaknesses in this category are typically introduced during the configuration of the software.

CWE-189: *Numeric Errors*

Weaknesses in this category are related to improper calculation or conversion of numbers.

CWE-20: *Improper Input Validation*

The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program.

CWE-200: *Information Exposure*

An information exposure is the intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information.

CWE-22: *Path Traversal*

The software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.

CWE-255: *Credentials Management*

Weaknesses in this category are related to the management of credentials.

CWE-264: *Access Controls*

Weaknesses in this category are related to the management of permissions, privileges, and other security features that are used to perform access control.

CWE-287: *Improper Authentication*

When an actor claims to have a given identity, the software does not prove or insufficiently proves that the claim is correct.

CWE-310: *Cryptographic Issues*

Weaknesses in this category are related to the use of cryptography.

CWE-352: *Cross-Site Request Forgery*

The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.

CWE-362: *Race Condition*

The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.

CWE-399: *Resource Management Errors*

Weaknesses in this category are related to improper management of system resources.

CWE-59: *Link Following*

The software attempts to access a file based on the filename, but it does not properly prevent that filename from identifying a link or shortcut that resolves to an unintended resource.

CWE-78: *OS Command Injection*

The software constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.

CWE-79: *Cross-site Scripting*

The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

CWE-89: *SQL Injection*

The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

CWE-94: *Code Injection*

The software constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment.

13 Appendix B: Stop Words

Appendix B contains the list of stop-words that was created for the purposes of the analysis described in Section 4.4.

a	ask	can	ends	gave	high	large
about	asked	cannot	enough	general	high	largely
above	asking	case	even	generally	higher	last
across	asks	cases	evenly	get	highest	later
after	at	certain	ever	gets	him	latest
again	away	certainly	every	give	himself	least
against	back	clear	everybody	given	his	less
all	backed	clearly	everyone	gives	how	let
almost	backing	come	everything	go	however	lets
alone	backs	could	everywhere	going	if	like
along	be	did	face	good	important	likely
already	became	differ	faces	goods	in	long
also	because	different	fact	got	interest	longer
although	become	differently	facts	great	interested	longest
always	becomes	do	far	greater	interesting	made
among	been	does	felt	greatest	interests	make
an	before	done	few	group	into	making
and	began	down	find	grouped	is	man
another	behind	down	finds	grouping	it	many
any	being	downed	first	groups	its	may
anybody	beings	downing	for	had	itself	me
anyone	best	downs	four	has	just	member
anything	better	during	from	have	keep	members
anywhere	between	each	full	having	keeps	men
are	big	early	fully	he	kind	might
area	both	either	further	her	knew	more
areas	but	end	furthered	here	know	most
around	by	ended	furthering	herself	known	mostly
as	came	ending	furthers	high	knows	mr

mrs	older	pointed	seeming	taken	turn	whether
much	oldest	pointing	seems	than	turned	which
must	on	points	sees	that	turning	while
my	once	possible	several	the	turns	who
myself	one	present	shall	their	two	whole
necessary	only	presented	she	them	under	whose
need	open	presenting	should	then	until	why
needed	opened	presents	show	there	up	will
needing	opening	problem	showed	therefore	upon	with
needs	opens	problems	showing	these	us	within
never	or	put	shows	they	use	without
new	order	puts	side	thing	used	work
newer	ordered	quite	sides	things	uses	worked
newest	ordering	rather	since	think	very	working
next	orders	really	small	thinks	want	works
no	other	right	smaller	this	wanted	would
nobody	others	right	smallest	those	wanting	year
non	our	room	so	though	wants	years
noone	out	rooms	some	thought	was	yet
not	over	said	somebody	thoughts	way	you
nothing	part	same	someone	three	ways	young
now	parted	saw	something	through	we	younger
nowhere	parting	say	somewhere	thus	well	youngest
number	parts	says	state	to	wells	your
numbers	per	second	states	today	went	yours
of	perhaps	seconds	still	together	were	
off	place	see	such	too	what	
often	places	seem	sure	took	when	
old	point	seemed	take	toward	where	

14 Appendix C: Classification Configuration

Appendix C contains the configuration for the Support Vector Machine algorithm with a cubic kernel used for machine learning in the last experiment of the analysis described in Section 4.4.

```
<?xml version="1.0"?>
<ML-CONFIG>
  <VERBOSITY level="1"/>
  <SURROUND value="false"/>
  <IS-LABEL-UPDATABLE value="true"/>

  <IS-NLPFEATURELIST-UPDATABLE value="true"/>

  <PARAMETER name="thresholdProbabilityEntity" value="0.2"/>
  <PARAMETER name="thresholdProbabilityBoundary" value="0.42"/>
  <PARAMETER name="thresholdProbabilityClassification" value="0.8"/>

  <multiClassification2Binary method="one-vs-others" numberOfThreads="10"/>
  <EVALUATION method="split" runs="3" ratio="0.8"/>
  <FILTERING ratio="0.0" dis="near"/>

  <ENGINE nickname="SVM" implementationName="SVMLibSvmJava"
       options ="-t 1 -c 0.7 -tau 0.4"/>

  <DATASET>
    <INSTANCE-TYPE>Text</INSTANCE-TYPE>
    <NGRAM>
      <NAME>Keygram</NAME>
      <NUMBER>1</NUMBER>
      <CONSNUM>1</CONSNUM>
      <CONS-1>
        <TYPE>Token</TYPE>
        <FEATURE>string</FEATURE>
      </CONS-1>
    </NGRAM>

    <ATTRIBUTE>
      <NAME>Class</NAME>
      <SEMTYPE>NOMINAL</SEMTYPE>
      <TYPE>Text</TYPE>
      <FEATURE>class</FEATURE>
      <POSITION>0</POSITION>
      <CLASS/>
    </ATTRIBUTE>
  </DATASET>
</ML-CONFIG>
```

15 Appendix D: Key Information Extraction Configuration

Appendix D contains the configuration for the Support Vector Machine algorithm with a cubic kernel used for machine learning in the analysis described in Section 4.5.

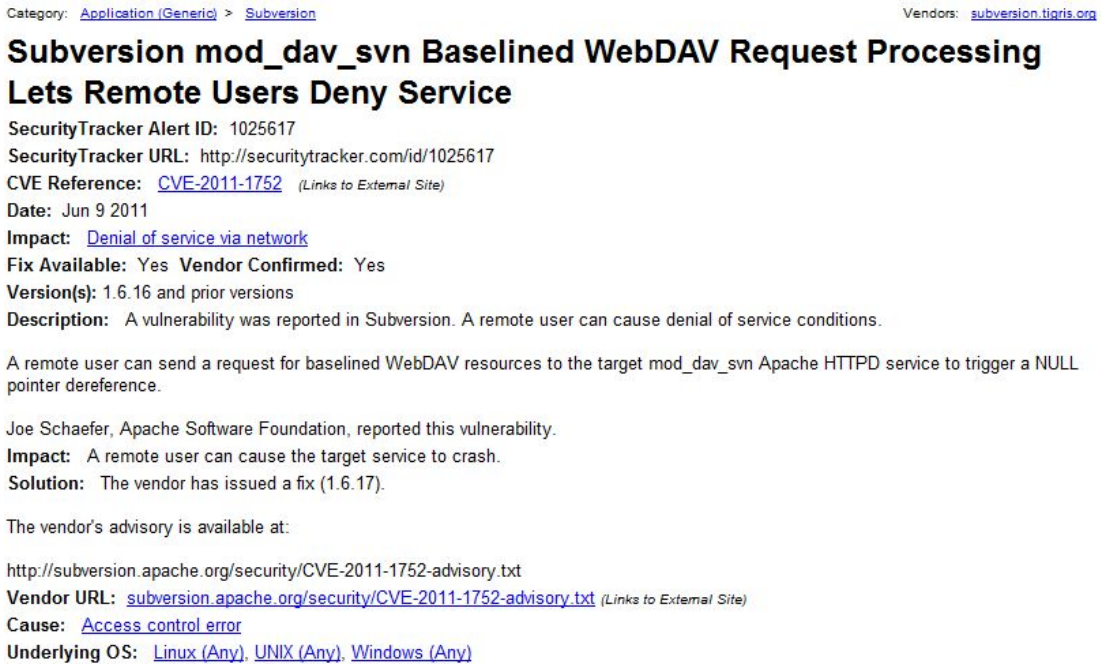
```
<?xml version="1.0"?>
<ML-CONFIG>
  <SURROUND value="true"/>
  <FILTERING ratio="0.1" dis="near"/>

  <PARAMETER name="thresholdProbabilityEntity" value="0.2"/>
  <PARAMETER name="thresholdProbabilityBoundary" value="0.4"/>
  <PARAMETER name="thresholdProbabilityClassification" value="0.5"/>

  <multiClassification2Binary method="one-vs-others"/>

  <ENGINE nickname="SVM" implementationName="SVMLibSvmJava"
    options="-t 1 -c 0.7 -tau 0.4"/>

  <DATASET>
    <INSTANCE-TYPE>Token</INSTANCE-TYPE>

    <ATTRIBUTELIST>
      <NAME>Orthography</NAME>
      <SEMTYPE>NOMINAL</SEMTYPE>
      <TYPE>Token</TYPE>
      <FEATURE>orth</FEATURE>
      <RANGE from="-5" to="5"/>
    </ATTRIBUTELIST>

    <ATTRIBUTELIST>
      <NAME>POS</NAME>
      <SEMTYPE>NOMINAL</SEMTYPE>
      <TYPE>Token</TYPE>
      <FEATURE>category</FEATURE>
      <RANGE from="-5" to="5"/>
    </ATTRIBUTELIST>

    <ATTRIBUTELIST>
      <NAME>Morpho</NAME>
      <SEMTYPE>NOMINAL</SEMTYPE>
      <TYPE>Token</TYPE>
      <FEATURE>root</FEATURE>
      <RANGE from="-5" to="5"/>
    </ATTRIBUTELIST>

    <ATTRIBUTELIST>
      <NAME>TokenKind</NAME>
      <SEMTYPE>NOMINAL</SEMTYPE>
      <TYPE>Token</TYPE>
      <FEATURE>kind</FEATURE>
      <RANGE from="-5" to="5"/>
    </ATTRIBUTELIST>
```

```
    <ATTRIBUTE>
      <NAME>KeyInformation</NAME>
      <SEMTYPE>NOMINAL</SEMTYPE>
      <TYPE>Mention</TYPE>
      <FEATURE>class</FEATURE>
      <POSITION>0</POSITION>
      <CLASS/>
    </ATTRIBUTE>
  </DATASET>
</ML-CONFIG>
```

16 Appendix E: Information Gathering Example

Appendix E demonstrates the process of information gathering described in Chapter 5 on illustrative examples.

Figure E.1 shows an advisory presented on the Security Tracker website.



Category: Application (Generic) > Subversion Vendors: subversion.tigris.org

Subversion mod_dav_svn Baselined WebDAV Request Processing Lets Remote Users Deny Service

SecurityTracker Alert ID: 1025617
SecurityTracker URL: http://securitytracker.com/id/1025617
CVE Reference: CVE-2011-1752 *(Links to External Site)*
Date: Jun 9 2011
Impact: Denial of service via network
Fix Available: Yes **Vendor Confirmed:** Yes
Version(s): 1.6.16 and prior versions
Description: A vulnerability was reported in Subversion. A remote user can cause denial of service conditions.

A remote user can send a request for baselined WebDAV resources to the target mod_dav_svn Apache HTTPD service to trigger a NULL pointer dereference.

Joe Schaefer, Apache Software Foundation, reported this vulnerability.
Impact: A remote user can cause the target service to crash.
Solution: The vendor has issued a fix (1.6.17).

The vendor's advisory is available at:

http://subversion.apache.org/security/CVE-2011-1752-advisory.txt
Vendor URL: subversion.apache.org/security/CVE-2011-1752-advisory.txt *(Links to External Site)*
Cause: Access control error
Underlying OS: Linux (Any), UNIX (Any), Windows (Any)

Figure E.1: *Security Tracker Advisory number 1025617*

During the process of advisory update the Advisory Updaters tool extracts the information directly from the web page and transforms it into the common format. The result for this particular advisory looks as follows:

```
<Advisory>
  <Source>
    <SourceName>SecurityTracker</SourceName>
    <OriginalAdvID>1025617</OriginalAdvID>
    <Link>http://securitytracker.com/id/1025617</Link>
  </Source>
  <Title>Subversion mod_dav_svn Baselined WebDAV Request Processing Lets
Remote Users Deny Service</Title>
  <ReleaseDate>2011-06-09</ReleaseDate>
  <Risk />
  <Summary>Access control error; Denial of service via network; A
vulnerability was reported in Subversion. A remote user can cause denial of
service conditions.A remote user can cause the target service to
crash.</Summary>
  <Solution>The vendor has issued a fix (1.6.17).</Solution>
  <AffectedProducts>
    <Product>Subversion - 1.6.16 and prior versions</Product>
  </AffectedProducts>
  <References>
```

```
    <Reference Type="Web">http://subversion.apache.org/security/CVE-2011-1752-
advisory.txt</Reference>
    <Reference Type="CVE">CVE-2011-1752</Reference>
  </References>
  <Credits />
  <BaseCVSS>
    <AccessVector />
    <AccessComplexity />
    <Authentication />
    <ConfidentialityImpact />
    <IntegrityImpact />
    <AvailabilityImpact />
    <BaseScore>0</BaseScore>
  </BaseCVSS>
</Advisory>
```

Another example shows an advisory from the National Vulnerability Database in its XML form:

```
<entry id="CVE-2006-2439">
  <vuln:vulnerable-configuration id="http://nvd.nist.gov/">
    <cpe-lang:logical-test negate="false" operator="OR">
      <cpe-lang:fact-ref name="cpe:/a:zipcentral:zipcentral:4.01" />
    </cpe-lang:logical-test>
  </vuln:vulnerable-configuration>
  <vuln:vulnerable-software-list>
    <vuln:product>cpe:/a:zipcentral:zipcentral:4.01</vuln:product>
  </vuln:vulnerable-software-list>
  <vuln:cve-id>CVE-2006-2439</vuln:cve-id>
  <vuln:published-datetime>2006-06-01T06:02:00.000-04:00</vuln:published-
datetime>
  <vuln:last-modified-datetime>2011-07-28T00:00:00.000-04:00</vuln:last-
modified-datetime>
  <vuln:cvss>
    <cvss:base_metrics>
      <cvss:score>7.6</cvss:score>
      <cvss:access-vector>NETWORK</cvss:access-vector>
      <cvss:access-complexity>HIGH</cvss:access-complexity>
      <cvss:authentication>NONE</cvss:authentication>
      <cvss:confidentiality-impact>COMPLETE</cvss:confidentiality-impact>
      <cvss:integrity-impact>COMPLETE</cvss:integrity-impact>
      <cvss:availability-impact>COMPLETE</cvss:availability-impact>
      <cvss:source>http://nvd.nist.gov</cvss:source>
      <cvss:generated-on-datetime>2006-06-01T12:55:00.000-
04:00</cvss:generated-on-datetime>
    </cvss:base_metrics>
  </vuln:cvss>
  <vuln:security-protection>ALLOWS_ADMIN_ACCESS</vuln:security-protection>
  <vuln:cwe id="CWE-119" />
  <vuln:references xml:lang="en" reference_type="UNKNOWN">
    <vuln:source>XF</vuln:source>
    <vuln:reference href="http://xforce.iss.net/xforce/xfdb/26737"
xml:lang="en">zipcentral-zip-filename-bo(26737)</vuln:reference>
  </vuln:references>
  <vuln:references xml:lang="en" reference_type="VENDOR_ADVISORY">
    <vuln:source>VUPEN</vuln:source>
    <vuln:reference href="http://www.vupen.com/english/advisories/2006/2049"
xml:lang="en">ADV-2006-2049</vuln:reference>
  </vuln:references>
```

```
  <vuln:references xml:lang="en" reference_type="UNKNOWN">
    <vuln:source>BID</vuln:source>
    <vuln:reference href="http://www.securityfocus.com/bid/18160"
xml:lang="en">18160</vuln:reference>
  </vuln:references>
  <vuln:references xml:lang="en" reference_type="UNKNOWN">
    <vuln:source>BUGTRAQ</vuln:source>
    <vuln:reference
href="http://www.securityfocus.com/archive/1/archive/1/435416/100/0/threaded"
xml:lang="en">20060531 Secunia Research: ZipCentral ZIP File Handling Buffer
OverflowVulnerability</vuln:reference>
  </vuln:references>
  <vuln:references xml:lang="en" reference_type="UNKNOWN">
    <vuln:source>OSVDB</vuln:source>
    <vuln:reference href="http://www.osvdb.org/25830"
xml:lang="en">25830</vuln:reference>
  </vuln:references>
  <vuln:references xml:lang="en" reference_type="UNKNOWN">
    <vuln:source>SECTRACK</vuln:source>
    <vuln:reference href="http://securitytracker.com/id?1016176"
xml:lang="en">1016176</vuln:reference>
  </vuln:references>
  <vuln:references xml:lang="en" reference_type="VENDOR_ADVISORY">
    <vuln:source>MISC</vuln:source>
    <vuln:reference href="http://secunia.com/secunia_research/2006-
35/advisory/" xml:lang="en">http://secunia.com/secunia_research/2006-
35/advisory/</vuln:reference>
  </vuln:references>
  <vuln:references xml:lang="en" reference_type="VENDOR_ADVISORY">
    <vuln:source>SECUNIA</vuln:source>
    <vuln:reference href="http://secunia.com/advisories/20179"
xml:lang="en">20179</vuln:reference>
  </vuln:references>
  <vuln:summary>Stack-based buffer overflow in ZipCentral 4.01 allows remote
user-assisted attackers to execute arbitrary code via a ZIP archive containing
a long filename.</vuln:summary>
</entry>
```

The result of the transformation of this particular advisory into the common format looks as follows:

```
<Advisory>
  <Source>
    <SourceName>NVD</SourceName>
    <OriginalAdvID>CVE-2006-2439</OriginalAdvID>
    <Link>http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-2439</Link>
  </Source>
  <Title>Stack-based buffer overflow in ZipCentral 4.01 allows remote user-
assisted attackers to execute arb...</Title>
  <ReleaseDate>2011-07-28</ReleaseDate>
  <Risk>High</Risk>
  <Summary>Stack-based buffer overflow in ZipCentral 4.01 allows remote user-
assisted attackers to execute arbitrary code via a ZIP archive containing a
long filename.</Summary>
  <Solution />
  <AffectedProducts>
    <Product>cpe:/a:zipcentral:zipcentral:4.01</Product>
  </AffectedProducts>
  <References>
```

```
    <Reference Type="Web">http://xforce.iss.net/xforce/xfdb/26737</Reference>
    <Reference
Type="Web">http://www.vupen.com/english/advisories/2006/2049</Reference>
    <Reference Type="Web">http://www.securityfocus.com/bid/18160</Reference>
    <Reference
Type="Web">http://www.securityfocus.com/archive/1/archive/1/435416/100/0/threa
ded</Reference>
    <Reference Type="Web">http://www.osvdb.org/25830</Reference>
    <Reference Type="Web">http://securitytracker.com/id?1016176</Reference>
    <Reference Type="Web">http://secunia.com/secunia_research/2006-
35/advisory/</Reference>
    <Reference Type="Web">http://secunia.com/advisories/20179</Reference>
    <Reference Type="CVE">CVE-2006-2439</Reference>
  </References>
  <Credits />
  <BaseCVSS>
    <AccessVector>NETWORK</AccessVector>
    <AccessComplexity>HIGH</AccessComplexity>
    <Authentication>NONE</Authentication>
    <ConfidentialityImpact>COMPLETE</ConfidentialityImpact>
    <IntegrityImpact>COMPLETE</IntegrityImpact>
    <AvailabilityImpact>COMPLETE</AvailabilityImpact>
    <BaseScore>7.6</BaseScore>
  </BaseCVSS>
</Advisory>
```

17 Appendix F: Contents of the Enclosed CD

The implementations described in Chapter 5 and Chapter 6 as well as the configurations and resources created for the purposes of the analysis described in Chapter 4 can be found on the CD attached to the thesis. The content of the CD is organized as follows:

- **analysis –** The folder contains the GATE related configurations and resources that were created during the analysis described in Chapter 4. The content is divided into two sub-folders:
  - **classification –** The sub-folder contains the resources related to the analysis described in Section 4.4.
  - **information_extraction** – The sub-folder contains the resources related to the analysis described in Section 4.5.
- **distribution –** The folder contains the distribution of the Vulnerability Management Tool application described in Chapter 6. The distribution also includes the Advisory Updaters tool described in Chapter 5. The folder can be simply copied to a hard-drive and the application can be then started via `VMT.jar`. The Advisory Updaters tool can be started via `UpdateAdvisories.bat`.
- **source_codes –** The folder contains the source codes of the applications implemented as part of this work. The content is divided into following sub-folders:
  - **VMT –** The Netbeans projects that form the Vulnerability Management Tool application. The projects were developed in NetBeans IDE 6.9.1.
  - **AdvisoryUpdaters –** The Visual Studio Solution of the Advisory Updaters application. The Solution was developed in Visual Studio 2010.
  - **ExtractCWE –** A small C# application that extracts description texts together with CWE categories from NVD advisories. The application was developed in Visual Studio 2010.
- **thesis –** The folder contains the electronic version of this thesis.
- **xml –** The folder contains various XSLT transformations and XML schemata that were created as part of this work.