Detecting known host security flaws over a network connection

Martin Andersson
Abstract

To test if a host contains any known security flaws over a network connection a Vulnerability Assessment (VA) could be made. This thesis describes different techniques used by VA tools over a network connection to detect known security flaws. To decrease the risk of flaws not being detected, several VA tools could be used.

There is no common way of merging information from different VA tools. Therefore the Vulnerability Assessment Information Handler (VAIH) has been developed. The VAIH system consists of three parts. First, a intermediate language format defined in XML. Second, modules that converts the output of VA tools to the intermediate language format. Third, a program for reading and displaying the intermediate language format.

The VAIH system makes it possible to merge the results from vulnerability assessment tools into one file that can be displayed and edited through a GUI.

Key-words: Vulnerability assessment, computer security, host security, network security, detecting security flaws.

Acknowledgments

I would like to start thanking iXsecurity for providing this great idea and supporting me with a contact person. I would further like to thank Johannes Gumbel, a security expert, good friend and contact person at iXsecurity who has inspired me. I would also like to thank Ola Flygt at Växjö University who has guided me through this thesis.
## Contents

1 Problem description  
   1.1 Background .................................................. 1  
   1.2 Motivation .................................................. 1  
   1.3 Purpose .................................................... 1  
   1.4 Target group ............................................... 1  
   1.5 Problem discussion ......................................... 1  
   1.6 Delimitation ............................................... 2  

2 Theory  
   2.1 Security flaws ............................................ 3  
   2.2 Discovering security flaws in general .................. 3  
   2.3 Network protocols ......................................... 5  
   2.4 Port scanning techniques ................................. 6  
   2.5 Version detection techniques ............................ 8  
   2.6 Operating System fingerprinting ......................... 9  
   2.7 Vulnerability Assessment ................................ 10  

3 Result  
   3.1 Problem solution ......................................... 12  
   3.2 VAIH requirements ....................................... 12  
   3.3 VAIH design ............................................... 13  
   3.4 VAIH implementation .................................... 17  
   3.5 VAIH tests ................................................. 20  

4 Discussion  
   4.1 Implementation discussion .............................. 23  
   4.2 Goals ..................................................... 23  
   4.3 Future work ............................................... 24  

References ....................................................... 25  

A Example VAIH module .......................................... 26  

B Intermediate Language format ................................ 28
1 Problem description

This chapter gives some background information in order to motivate why this thesis has been written. This chapter also establish the purpose of this thesis.

1.1 Background

Security is a big issue for many organizations today. There exists many threats to a big organization and some level of security in the organizations computer network are usually required. Servers containing sensitive information might only accept connections from hosts in the local network. Therefore the security of each individual host becomes very important, since the host itself could be used to launch further attacks. That can also be the case when the host itself contains sensitive information that needs to be protected.

To bring some confidence about the computer network within an organization a vulnerability assessment of the hosts could be made. The vulnerability assessment is made as a test to discover security flaws within the hosts in the network, if host flaws are not fixed an attacker might try to use them in his favor.

The problem of discovering host security flaws are two folded. First, information is needed about each host we want to examine. This information can tell what security flaws a host might contain. Secondly, there is a need to manage the information collected. In a big organization with hundreds, maybe thousand of computers it is not trivial how to manage all the information that is collected.

1.2 Motivation

Studies, such as the one performed by J. Snyder [2], has shown the need of a system that can enhance vulnerability assessment by combining information from different vulnerability assessment tools, or sometimes combining previously performed assessment results with new assessment results from the same tool.

The company iXsecurity who proposed this subject has also described that handling large quantities of information is a problem when performing vulnerability assessment. Some system is needed to handle, and give a better overview of the process then handling each file of information separately.

1.3 Purpose

The purpose of this thesis is to give the reader knowledge of how a vulnerability assessment can be done by first collecting information and then handling the information. The purpose is also to derive a better method for handling the information collected then parsing each information source manually to get an overview of the information.

1.4 Target group

This thesis is for anyone in the computer science area who are interested in how known security flaws can be discovered on hosts using a network connection, and how the information from different assessment can be combined into one information source.

1.5 Problem discussion

To estimate what security flaws lays within hosts of a specific network a method is needed for finding the flaws that causes threats to the hosts. This thesis aims towards describing
how security flaws can be discovered and also how to handle the information about flaws detected. The problem formulation is therefore:

How can known security flaws be discovered on remote hosts over a network connection, and how can all the information be handled?

1.6 Delimitation

This thesis is only concerned with how known host security flaws can be detected over a network connection and how the information about the hosts can be handled to ease the task of handling large and multiple information sources.

This thesis aims towards building a system for the task of information handling. This thesis is a Bachelor level thesis, meaning that ten weeks are dedicated for development and thesis writing. Therefore the system has to be a small scale system that is possible to implement during this time. iXsecurity has requested that the system should work under Microsoft Windows XP, it is also desirable if it works on other platforms, but not required.
2 Theory

The sections throughout this chapter explains what a security flaw is and how a security flaw can be detected over a network connection. This chapter tries to give the reader enough information to understand what security flaws are and how they can be detected.

2.1 Security flaws

There exists several kinds of security flaws, this thesis is not intended to extensively cover all kinds of security flaws. Since the term vulnerability or security flaw are used throughout this thesis a general explanation is required.

One definition used to describe the term vulnerability is stated in the book the Shellcoder's Handbook [1] as follows:

A flaw in a system's security that can lead to an attacker utilizing the system in a manner other than that which the designer intended. This can include impacting the availability of the system, elevating access privileges to an unintended level, complete control of the system by an unauthorized party, and many other possibilities.

Also known as a security flaw or security bug.

The definition of a vulnerability depends highly on the context. A vulnerability does not have to be in the area of computer security. The definition stated above is taken from the context of computer software vulnerabilities, which is the vulnerability area of this thesis.

2.2 Discovering security flaws in general

A big threat for a host connected to a network is that someone might want to gain illegal access to the host. There are many reasons of why computer systems are broken into, some of the reasons might be to gain sensitive information stored in a system, to use the system for further attacks or maybe just to prove that it’s possible.

Discovered security flaws are posted daily within different security communities. It is therefore an important task for every administrator too keep the hosts in the network up to date. Even someone who is not an expert can download source code that exploits these posted security flaws. Some of the source code for exploiting security flaws were written as a proof that the security flaw do exist, these proofs can sometimes be used by an attacker to gain illegal access to a host. In this sense both the administrator and the potential attackers face the same problem: how to know whether a specific host are vulnerable or not?

To answer the question two phases needs to be completed:

1. Collect information about the hosts of interest.

2. Gather and analyze the collected information.

After performing the two above mentioned steps the analyzer of the host in the network should have a good foundation for deciding what actions should be taken in order to increase the host security.
2.2.1 Collect information about the hosts of interest

The first phase includes getting information about interesting hosts. This phase may include huge numbers of hosts. An administrator might try to collect information from every host in his network, for example, a big corporations hosts. A potential attacker might try to gain unauthorized access to hosts for launching further attacks, in that case he might try to gain information from as many hosts as possible on the internet.

The collecting phase can be divided into sub-phases:

1. Determine what services a host is running.
2. Estimate/determine what version of the identified service is running.
3. Determine whether the discovered version is vulnerable.

Phase 1, determine what services a host is running, can be performed by launching a port scan on the target host. Port scanning techniques are described in detail later in this thesis and requires a deep knowledge of network protocols.

Phase 2, estimate/determine what version of the identified service is running, can usually be performed similar to port scanning but these techniques has to be more tweaked to fit a single service. This phase will also be described in more detail later on.

Phase 3, determine whether the discovered version is vulnerable, can be established by using a database of known vulnerable versions. This thesis only considers known security flaws, techniques for finding new security flaws will not be described.

2.2.2 Gather and analyze the collected information

A vulnerability assessment (VA) tool usually implement some interface to present the results of the VA performed. Such an interface may even include exhaustive analysis featuring diagrams and other presentations. This thesis aims towards providing a method for displaying the information of which security flaws a given host in the network contains. For this type of analyzes it is enough to be able to search or filter on some specific vulnerabilities and to easily get host information, such as an IP address.

Even if a single VA tool features great gather information and analyze capabilities one VA tool might not be enough.

Joel Snyder presented an article [2] that tested five popular VA tools on an known environment. The test showed that the VA tools are far from flawless. Different VA tools are better then others, but these tools might get confused under certain circumstances and make mistakes. On gathering the information Joel Snyder writes in his article [2] that

Data management is vital, especially if you plan to scan your network more than once. It’s important to be able to see results in many different formats, pivoting across different views, and comparing data over time. We were disappointed in the functionality we found.

Similar conclusions were determined in a paper by Debra Beck, Rohan Amin, Eric Hutchins, Michael Poddo [3] where they wrote

To this end, most scanning tools offer little, often producing statically structured reports with no means of designating the criticality of systems or even filtering false positives. In addition, VA tools typically store the results of each scan separately, hindering the opportunity for useful historical analysis.
Previously performed studies shows that in some cases, as mentioned above, there might be good presentations of a single vulnerability scan. But many tools lack the functionality of being able to shown multiple scans. It would bring more confidence of the assessment if more then one VA tool were used. Trying to merge large log files by hand from different VA tools in an attempt to combine the information can be chaotic and may take long time. Therefore a method for dealing with the information produced by VA tools are needed.

2.3 Network protocols

To detect security flaws on a host over a network some test are performed over the network. To understand how these tests works a basic understanding of computer networks is required. This section is not ment to give an exhaustive explanation about network communication, it is rather a short overview to remind the reader of how it works.

In order to explain how different protocol suits are related the TCP/IP reference model (figure 2.1) is frequently used. The protocols briefly explained in this thesis belongs in the internet and transport layers.

Each layer in the TCP/IP reference model is encapsulated within the lower layer when network communication takes place. That is, the application layer is encapsulated within the transport layer and so on. All the way down to the physical layer. Each layer adding its header (some layer specific data) to the data that is to be transfered. This allows for the receiver to strip the headers, layer by layer, when the package is received. Thereby regaining the data on the receiving side.

Network and internet layers are usually implemented by the operating systems, we will later see that the differences between different implementations of the layers can help us gain information about hosts.

![TCP/IP reference model](image)

Figure 2.1: TCP/IP reference model

2.3.1 Internet Protocol

The Internet Protocol (IP) is the protocol used to transfer data from one host to another host [4]. The IP provides support for dividing data into smaller pieces, called datagrams. It also provides IP addresses which helps to identify a host and a source. The IP layer does not consider which order the datagrams arrives on the host, this service is provided by higher layers.
2.3.2 Internet Control Message Protocol

The Internet Control Message Protocol (ICMP) is located within the network layer, just as the Internet Protocol [5]. For example, an ICMP message is returned when a host destination specified by a datagram could not be found. No ICMP messages are sent due to loss of a ICMP message.

2.3.3 Transmission Control Protocol

The Transmission Control Protocol (TCP) lays on top of the IP [6]. Which makes TCP a higher level protocol, contained within the transport layer in the TCP/IP reference model. TCP provides the feature of reliable network communication between two hosts. This is done by first connecting to the remote host, using the 3-way handshake, and then verifying each packet received. TCP also makes sure that each packet received is in the order the sender sent it.

In figure 2.2 the TCP header is shown.

![TCP header diagram](image)

Figure 2.2: TCP header

2.3.4 User Datagram Protocol

The User Datagram Protocol (UDP) is also located on top of the IP [7]. This makes the UDP, just as TCP, part of the transport layer. UDP does not guarantee that packets are delivered, or that they are delivered in the order they were sent. These features makes UDP unreliable but don’t generate as much network traffic as TCP.

2.4 Port scanning techniques

Port scanning is a set of techniques used for detecting services that are running on a host machine. There are a few different techniques to perform a port scan described in this thesis.
2.4.1 TCP connect scan

One technique is to open a TCP connection on each port on the targeted host [8]. This technique makes a normal connection to each port on the target machine, if a SYN/ACK is responded it usually means that the port is listening [9]. An ACK packet is then sent to the target to establish the full connection required by the three-way-handshake.

2.4.2 TCP SYN scan

The TCP SYN scan technique is also based on the TCP protocol three-way-handshake. It is performed by first sending a packet, to the target host and port, with the SYN-flag set in the TCP header. If the port is open the target responds by sending a TCP packet with the SYN- and ACK-flag set. If the port is closed a TCP packet with RST-flag set is replied. In a normal three-way-handshake (when we actually tries to establish a connection) a last ACK-packet is replied when a SYN/ACK-packet is received to establish a connection. When SYN scanning a target the last step of sending an ACK-packet is replaced with an RST packet, a full connection is never established [9].

The TCP SYN scanning technique is not detectable by targets that only logs full TCP connections. However, "failed attempts" could be logged as well.

2.4.3 TCP NULL, FIN and Xmas scan

Implementations that follows the TCP RFC [6] will return a RST packet when a packet that does not have SYN, ACK or RST flag is sent to a closed port, if the port is open there will be no response [8]. The TCP null scan achieves its name by not setting any flags in the TCP header, FIN scan by setting the FIN flag, Xmas scan by setting the FIN, PSH and URG flags making the header lit up like a Christmas tree.

If the implementation does not respond as it should, that information can be used in discovering what system the target host is running.

2.4.4 UDP scan

The big difference between scanning UDP and TCP is that TCP is a connection oriented protocol, UDP is not. This means that a UDP port that is open doesn’t have to send an ACK packet if it is open, it also means that a error message is not required when the port is closed [10].

Some hosts reply with a ICMP_HOST_UNREACH when the port is closed. When this is the case it is possible, by elimination, to conclude which ports are open.
However, there can be a real performance penalty by using UDP scanning since some hosts restrict the maximum ICMP error messages, which means that delay in the scan has to be added. Also we have to scan a port that is not responding several times to really conclude that it is open, since the packet might as well be lost.

2.5 Version detection techniques

When a port is discovered through port scanning a service has been detected. The question to answer next is: What service has been detected?

A naive method for answering this question would be to assume the standard port number. This method is naive because a service could be running on any port which would confuse a scanner assuming standard port number. Even if the service is actually using its standard port it’s not a hole lot of information. It would be nice to know what version of the service is running in order to know if there is any security flaws to exploit [11].

2.5.1 Banner version detection

When a port has an associated TCP service one way to investigate it further is simply to connect to the service. Some services announce the current version they are running when a TCP connection are established [11].

The method of investigating the data sent after a TCP connection is established has a similar flaw as the method of assuming standard port numbers. Since a service could send a fake banner to confuse the version detection. Even if the scenario of faking the banner is not as usual as using a none standard port number it is not very reliable.

2.5.2 Detecting version through probing

Probing to detect the version is the technique of sending data that contains data the service should respond to [11]. For official protocols the protocol description can be found which defines what data should be replied when certain data is received. New versions obviously change something, a lot of times these changes can be recognized through probing. Some versions may crash on certain data, some versions contains new functionality and older versions may then not respond or respond with error messages when that functionality, which were not yet invented, are called.

Since services vary in their implementation there is no universal method for detected arbitrary versions on arbitrary services. Each service is more or less unique and therefore has to be treated separately. Different tools uses different tests to determine a specific version, for example: the Nessus scanner [12] has each and every test written as a plugin. This approach means that you can download plugins written by other, modify them yourself and so on. Similar approaches are taken by other tools that has to detect a specific version in that each version test is performed separately.

By making each test a separately code snippet that the one running the test may modify allows a selection of aggressiveness. For example, an administrator that are running a test on his LAN may not want to perform tests that crashes a service on certain input data. This may effect the business and cause a lot of harm. Likewise, an attacker may not want to crash a service so that the administrator discovers the flaw himself while trying to figure out what happened. By allowing each test to be modified such tests can be disabled and instead investigated manually.
2.5.3 Detecting version through listening

A method similar to version probing can also be used. The big difference is that instead of sending information to detect the correct version, listening is used. A network sniffer, a piece of software used to listen to traffic that occurs on the network, can be used. Information can then be collected about a given service and thus the version can be determined in the same way as if the information was given through responses of probing. This method requires that network traffic from the service, and perhaps the right traffic, is generated from the service. This means that it is sometimes a slow technique that has to wait for information exchange to occur.

2.6 Operating System fingerprinting

Operating System (OS) fingerprinting is a set of techniques to determine what OS a remote host is running. This information might be very valuable since some OS versions contain security flaws. Knowing the correct OS may also help when a version detecting service is running by eliminating versions that couldn’t possibly be running on the current detected OS [13]. To know what OS a host is running may also be required to be able to exploit a security flaw, which makes such information vital for an attacker.

2.6.1 Probing

The technique of sending probe packets to investigate what OS the target is running is very similar to probing a service for version detection. The biggest difference is that several types of packages are sent to the remote host. For example, we may send TCP, UDP and ICMP packages to the remote machine with various header settings [13].

2.6.2 Passive fingerprinting

Passive fingerprinting is the technique of determining the OS on a specific host by sniffing the network [13]. This is actually very similar to the probing technique described earlier. In both cases the response on a certain packet is investigated. The passive technique are more complicated because we can not choose what to look for. In probing we decide how a packet should look like and then we jump to conclusions based on the response. When sniffing the sent packet can not be controlled, this may lead to longer identification time. If the investigation is time limited it might lead to an inconclusive result because enough conclusions could not be made by the traffic that was available.

2.6.3 Exploiting

Another way to detect a OS, or a specific version of an OS is to try exploiting security flaws [13]. This technique is best used when an OS has been detected but the exact version is still not determined. The reason is that security flaws are often exploited in different ways depending on the OS. By launching a successful attack we have determined that the version is one that is vulnerable to the specific attack, and if the attack fails we can draw the conclusion that the version is not one of the affected by the security flaw.

2.6.4 Responses

Different OS may implement different response times. When using the TCP certain packets requires certain responses to guarantee that there was no data loss. These times may
be measured and then compared, sometimes it will lead to an OS fingerprint. Although usually many probes of this kind are required due to network failures. Packets might get lost and produce false response times [13].

2.7 Vulnerability Assessment

The term Vulnerability Assessment is frequently used in the area of computer security. There are many different definitions of the term Vulnerability Assessment. The Wikipedia Encyclopedia [14] states that

Vulnerability Assessment (VA) is the process of identifying and quantifying vulnerabilities in a system. The system being studied could be a physical facility like a nuclear power plant, a computer system, or a larger system (for example the communications infrastructure or water infrastructure of a region).

In other words, VA can actually be used in wider ranges than computer networks. However, this thesis only covers VA of hosts within a network. In the context of network host security, VA is the process of identifying and quantifying host security flaws. The Wikipedia Encyclopedia [14] further states that

Assessments are typically performed according to the following steps:

1. Cataloging assets and capabilities (resources) in a system
2. Assigning quantifiable value and importance to the resources
3. Identifying the vulnerabilities or potential threats to each resource
4. Mitigating or eliminating the most serious vulnerabilities for the most valuable resources

These four steps can be applied to the host security area as well. The cataloging of resources can be done after gathering the information about different hosts in the network. This type of information is usually sufficient from a simple port scanner that can verify whether a host is online or not. The administrator of the network may also have information about the network resources at hand.

Assigning values based on importance can be done by the person performing the assessment. For instance, an server containing valuable customer information may be of greater importance then a regular workstation.

The identification of potential threats can be done by determining the version of a program running on a specific host. Known vulnerabilities is frequently posted within security communities. A VA tool can simply keep a database of all vulnerable versions and programs known today, in order to determine that a security hole exists.

Serious security holes has to be plugged in order to maintain a secure network. The process of eliminating security flaws is not covered in this thesis.

2.7.1 VA tools

A Vulnerability Assessment tool is a tool that assists the analyzer in his work of making the vulnerability assessment of a network. A VA tool usually contains three phases [2]

1. Identify hosts that are connected to the network.
2. Identify what services the connected hosts are running.


These tests can be performed by an advanced port scanner (that features version detection). But there are also more sophisticated tools that keeps track of how the version is vulnerable and how the vulnerability may be exploited. Some tools even has the possibility to launch attacks when a vulnerability is discovered in order to confirm its existence. By keeping track of it vulnerabilities, suggestions about fixing them can usually be provided as well.
3 Result

This chapter describes a method for dealing with the information collected about security flaws.

3.1 Problem solution

A problem that exists today is that VA tools might not be able to detect all known vulnerabilities. To increase the certainty of VA analyses a tool is needed to summarize the outcome of several tools. A tool that can manage multiple information sources would increase the efficiency of the VA analyses compared to investigating each log file, created by a VA tool, manually.

Another problem that has to be solved is that a VA tool might not support merging of information. The case might be that two analyzes of a network were made at different times, this might result in two unmergable sources of information [2].

Problems may also arise when large amounts of information is collected by tools that do not support some graphical user interface (GUI) to provide easy browsing through information. The amount of information might simply be so large that it is hard to get an overview.

To solve these problems I have designed and implemented a tool for collecting and managing the information from different VA tools. This system is called the Vulnerability Assessment Information Handler (VAIH).

3.2 VAIH requirements

This section tries to establish some of the requirements for the solution.

3.2.1 Core requirements

Solving the problems mentioned throughout this thesis can be stated as requirements for the solution. The requirements listed below are the core requirements, meaning that in order to solve the problems mentioned throughout this thesis the solution has to satisfy these core requirements.

1. Be able to collect information produced by different available tools for discovering security flaws.

2. Handle large amounts of information (large networks).

3. Display available information of interest.

Requirement 1 provides more confidence to a VA by providing information from different tools into the analyze. This requirement is also easy to measure whether it has been fulfilled by combining different information sources in a test.

Requirement 2 has to be achieved for the implementation to be useful in the real world. Since a corporation might use different vulnerability tools to collect information about their network containing thousands of hosts.

Requirement 3 is the foundation of the analyzes. There is no point in collecting information if the collected information can not be used. So there is no point in using a program that collects information that was not of interest to begin with. The problem with this requirement is that different occasions may requires different information. There are also no good way to measure whether the information selected is of interest or not within the time constraints of this thesis.
3.2.2 Optional requirements

Optional requirements should be fulfilled in as large extent as possible. However, failing to meet some of the optional requirements would not lead to a completely meaningless implementation, although it may affect the usability of the implementation.

1. Search and filter information.

2. Possibility to add additional information.

3. Ability to mark vulnerabilities as verified.

Requirement 1, were proposed by iXsecurity, has its roots in core requirement 2. Since the implementation are supposed to handle large amounts of information, it would be nice to have some functionality that can reduce the information for the user when looking for something specific.

Requirement 2, were proposed by iXsecurity, is substantiated in the fact that there is someone performing the assessment. Sometimes the assessment tool does not generate all information that is of importance to the analyzer. By providing functionality to add information the analyst does not need another information source, were he could keep his own notes. This requirement contributes to the core requirement 3.

VA tools sometimes come up with false positives vulnerabilities [2]. The tool might find something that it marks as a vulnerability, but when the flaw is investigated by the analyzer it is concluded that the flaw does not exist. In such case, the ability to mark a discovered vulnerability as false positive is important to exclude such information. Adding this information makes it possible to browse through positive vulnerabilities with the filter functionality mentioned above, which contributes to the core requirement 3. This functionality were also proposed by iXsecurity.

3.2.3 System requirements

iXsecurity has requested that the solution should work under Microsoft Windows XP. It’s also desirable if the solution works MAC OS X 10.4 and latest versions of Linux. Which is why other platforms can be seen as optional system requirements.

3.3 VAIH design

To achieve core requirement 1, modifiability is implied. Since the implementation itself can not consider all available tools for discovering security flaws, there are simply too many. Not only is there to many tools available, tools for discovering security flaws are constantly evolving. To implement support for each tool that will be used within a VA would mean that we also should predict future tools.

The aim has to be towards reshaping the information produced by any tool to fit an implementation, the implementation should of course consider what information is reasonable to include. The fact that new tools are invented and existing tools evolving implies that a lot of updates may have to take place in order to consider new available information.

Core requirement 2 need some stability of the system to hold. To handle large amounts of information (memory) means that the implementation would be highly affected by, for example, memory leakage.

Core requirement 3 is not an easy task. The information can’t be more then provided by different tools used to collect information, although several tools can be used. Some GUI also has to show this information, which might mean that the information has to be restricted in some way.
3.3.1 System design decisions

The summary of design goals to strive for is shown in table 3.1. The design goals should of course be fulfilled in as large extent as possible. The optional requirements are all very functional specific requirements, which makes them easy to achieve with an implementation.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect information from many different VA tools</td>
<td>Modifiability</td>
</tr>
<tr>
<td>Handle large amounts of information</td>
<td>Stability</td>
</tr>
<tr>
<td>Optional requirements</td>
<td>Functionality</td>
</tr>
</tbody>
</table>

Table 3.1: Property table

One way to achieve modifiability would be to interact with VA tools through dedicated modules. Making independent modules to interact with different VA tools provides a solution that does not affect the other parts of the system when the module is changed.

For example, one specific VA tool might not provide all the information one might wish for. The developers might discover such a lack of information and provide more complete information in later versions of the product. This change would only affect the module that has to absorb more information to complete its task.

In order to have independent modules the information that could be captured by the modules has to be defined. This is due to the Graphical User Interface (GUI) that is supposed to show the information. Some knowledge of the information is needed in order to present it in a suitable way. Also, a knowledge is needed in order to parse the information and use it in different contexts. These problems can be solved by restating the information collected in a well defined Intermediate Language (IL). The IL works as a bridge between the GUI and the independent modules. This would also allow for different GUI implementations, another implementation might be of interest when needs of presenting the information in other ways exists. It can also be the case when a new GUI is needed to adjust to new platforms.

The last parts of the system is a solution for handling and displaying the information. Since both handling the information and the GUI is highly dependent on the IL these can be developed together. If the IL should change, both the GUI and the handler (parser, objects...) needs to be updated in order to take the new information into account.

However, this solution may also restrict the usability of the system. If a vulnerability analyzer finds the information contained within the IL useless there is no easy change to the system that would make it useful.

3.3.2 System architecture

The complete system architecture is composed of three different sub-system: modules, intermediate language and a GUI/engine.

The VAIH module redefines the information collected from a certain tool to the intermediate language format. The information amount to collect from a certain tool is a decision left for the the module writer.

The Intermediate Language (IL) format defines what information the system can handle. If the assessment requires information not contained within the IL specification the VAIH system can not be used for the complete assessment.
The GUI/engine is responsible for handling and displaying the information contained in a IL file. Filter, searching, saving, modifying are all examples of handling the information. The GUI is responsible for the interaction with the user.

### 3.3.3 Module design decisions

Since the VAIH modules are independent, with the restriction that they have to produce information in the IL format, they can be designed as the developer see fits. However, a module were developed for testing purposes which may be used as a guideline for how VAIH modules may be developed.

The VAIH module were designed to use a complete library for reading information produced by a certain VA tool, as shown in figure 3.2. This principle can be used by using the same library, although it requires that the output format of the information produced by the target VA tool is the same. More information about specific formats are described in the implementation section.

### 3.3.4 Intermediate Language design decisions

The difficulty of designing the IL lies within the selection of information. There exists languages constructed for parts of the computer security area. One are is the Intrusion Detection Systems (IDS). There are different types and solutions of IDS, one type is a distributed IDS [15]. A distributed IDS uses a Intrusion Detection Exchange Format (IDEF) to enable communications between IDS. The Intrusion Detection Message Exchange Format (IDMEF) includes much more information then is required for the task of gathering VA tool output. For example, IDMEF includes specific information about different alerts of the system status [16]. One reason why the IDMEF is not applicable for network host security as well is that IDS works in a different perspective. An IDS module
can be running on a host, making it part of the host. Security aspects are not the same when comparing internal against network perspectives.

A lot of the information could of course be reused, such as host information (interface addresses), risk assessment and so on. The techniques described so far by this thesis is of course a good basis for the information that needs to be considered.

**Host information**

A host may have one or more network interfaces. The host may also contain vulnerabilities that can not be directly connected to any of the services provided by the host. Such vulnerabilities may be flaws in the OS.

**Interface information**

A interface has different types of addresses, hardware address (MAC address), IPv4 address and IPv6 address is the address types supported.

There may be open ports associated with the interface. These ports are open to provide some kind of service.

A specific network interface may be dedicated for a special task. For example, a host might have a dedicated network interface for a ftp server. This functionality is good to document since it may allow Denial of Service (DoS) attacks. The functionality of the interface may also be good to know in order to better understand how the network is built and configured.

IP addresses are sometimes associated with different names. These names are resolved by asking some kind service to resolve a given IP address. To get a better overview and understanding of a system, keeping track of a hosts different names and what service it is that are providing the name is desirable. Also keeping track of what name servers are resolving different names for a host may also be important for the organizational security. Since, for example, some service provided by the web can be denied by attacking the name server.

**Port information**

When a open port is found and documented it is important to know on what port number it was discovered. If any version detection was successful the service that was detected should also be included. All vulnerabilities that is known based on the service should also be included.

**Vulnerability information**

When a vulnerability is discovered and publicly announced, it is usually associated with some reference number. There exists many different databases over vulnerabilities that can be associated with different software versions. When a VA tool discovers a vulnerability it may include from which database the information about the vulnerability were collected and also provide a reference number. The reference number for a vulnerability may be different among databases, even though they might describe the same vulnerability. In order for the analyzer to be able to find more information about a discovered vulnerability this information has to be included.
If the analyzer finds out where the source of the vulnerability is located it should be documented. This information makes it easier to understand how the problem could be fixed.

In order for the analyzer to know whether a vulnerability has been confirmed as existing or not, the information about if a vulnerability has been marked as verified is required.

Finally, in order to provide information about the importance of a vulnerability a risk factor is assigned to each vulnerability. This factor can be used by the analyzer to give important vulnerabilities high risk status.

**Other information**

The IL format should also include information which has no given label. This information type might be used when the module developer wants to include something that was not covered by the IL. It might also be the case that the analyzer wants to add his own information, this should of course be possible.

### 3.3.5 GUI/Engine design decisions

The GUI and engine is located in the same part of the system, as described earlier. The design is shown in figure 3.3.

![Figure 3.3: The VAIH GUI/Engine](image)

The engine is responsible for parsing the IL format and storing the information locally while the program is running. The engine is also responsible for output in the IL format. Functionality such as searching and filtering are also handled by the engine.

The main goal of the Graphical User Interface (GUI) is to allow a user to easily change the information contained within a IL format file. This is done by interacting with the engine that stores the information during run-time. Since allot of the information contained about a host is supposed to be editable most of the information is displayed in text boxes and could be edited directly. It is also important to be able to get an overview of all the hosts found on a network and if they contain any vulnerabilities. This is done by showing a list of all known hosts on the left side (figure 3.4). Once a host is displayed a summarize of the vulnerabilities is displayed. Figure 3.4 shows a shows how a specific vulnerability is presented to the user.

### 3.4 VAIH implementation

This section describes implementation decisions that were taken in order to implement the design, described in the previous section. Decisions such as implementation language are included in this section. The exact implementation, such as objects, are not described in this thesis.
Figure 3.4: A vulnerability displayed.
3.4.1 Module implementation

VAIH modules responsibility is to read and write text. One programming language that has great text handling abilities is perl. The perl reference guide [17] states that

Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information.

This description is a great fit for the purpose of VAIH modules, which is why perl is selected as the VAIH module implementation language.

Appendix A provides a VAIH module as an example of how a module could be implemented. The module reads an Extensible Markup Language (XML) format that is the output of the Nessus scanner. In order to read XML files a standard perl module called XML::Simple is used. The XML::Simple module can be downloaded through the Comprehensive Perl Archive Network (CPAN) website [18].

3.4.2 Intermediate Language implementation

The IL could have been implemented as a new file format, however it would have required a parser inside the VAIH engine. To increase reuse of different modules in the VAIH system, that affects both the time required for development and stability of the system, Extensible Markup Language (XML) file format is chosen. Since XML is so widely used these days a number of parsers already exists in different programming languages.

The IL format can be found in Appendix B. The node, interface, vuln, vuln id and port information may be repeated to provide information about several instances.

3.4.3 Engine implementation

iXsecurity, who proposed the idea of implementing a system such as VAIH has requested that the system should work well on PC’s running Microsoft Windows XP operating system. However, it’s desirable if it works on PC’s running Linux or even MAC’s running OS X.

Besides good Windows support the chosen language should be a high level language. The reason is to enable reuse, and thereby increasing the stability, in as large extent as possible. This project is also under heavy of time pressure, which makes reuse a good idea.

One language that provides XML parser, easy Windows GUI creation and a lot of reuse is Microsoft’s own language, C-sharp using the .NET framework (sometimes abbreviated C#). There also exists compilers for system’s not using Windows. One such compiler is developed by the .gnu organization. On the .gnu website [19] the following statement can be found

DotGNU Portable.NET, an implementation of the Common Language Infrastructure (CLI), more commonly known as ".NET", includes everything that you need to compile and run C# and C applications that use the base class libraries, XML, and Systems.WinForms. Windows.Forms. Currently supported CPUs: x86, ppc, arm, parisc, s390, ia64, alpha, mips, sparc. Supported operating systems: GNU/Linux (on PCs, Sparc, iPAQ, Sharp Zaurus, PlayStation 2, Xbox,...), *BSD, Cygwin/Mingw32, Mac OS X, Solaris, AIX.

By using the System.WinForms.Forms library the requirements of the programming language is met by using C#.
3.5 VAIH tests

All tests were performed on MAC OS X version 10.4.7, Linux Slackware 10.2, Microsoft Windows XP SP2. These systems will be known as OS X, Linux and Windows, respectively, throughout this section.

Linux and Windows tests were performed on a computer with AMD 64 3200+ processor and 1GB of RAM. The MAC test were performed on a Powerbook with 1.5GHz PowerPC G4 processor and 512MB RAM.

3.5.1 Module test

The perl module provided in Appendix A has been tested successfully on all systems (mentioned above).

Two tests were performed using the Nessus scanner to provide information. The first test scanned one computer. The second test scanned two computers. In both cases the VAIH module were able to extract all of the expected information. The module were also successive in writing the information into a VAIH IL format file (the VAIH IL format can be found in Appendix B).

3.5.2 Engine test

The engine tests were performed on all systems (mentioned above) using 10, 100, 1000 and 10000 individual nodes. Each node containing one interface, one open port and two vulnerabilities. Different fields, such as comments, were filled with test values.

The complete test is said to be successful if the following test cases worked as expected:

- Import (read) test. Information from a VAIH IL format file (shown in Appendix B) should be read and presented for the user, containing as much information as provided by the file. Even if the XML file contains tags that are not specified by the VAIH IL format the reading should not halt or affect valid information. Tags that are not specified, and its content, should be ignored.

- Export (save) test. Some arbitrary nodes are selected and some information contained by these nodes are modified. The test succeed if the exported file is in the VAIH IL format (specified in Appendix B) and the information, including the modified information, is saved.

- Filter test. The only filter available in the VAIH engine is a filter that only shows vulnerabilities marked as Not Verified. The test is performed by marking arbitrary nodes as Not Verified. If only the nodes marked are shown when the filter is applied the test succeeds.

- Search IPv4 test. Each host in this test has one interface containing a unique IPv4 address. One arbitrary IPv4 address is selected and searched for. If the correct node is displayed the test succeeds.

- Search vulnerability id. In each host one vulnerability has a unique number, one such number is selected and searched for. If the correct node is displayed in a result list, the test succeeds.
Windows results
In table 3.2 the test results for Windows are shown. When using 10, 100 and 1000 nodes the implementation behaves as anticipated. However, when using 10000 it takes the test computer about 90 seconds to read the file and displaying the nodes. Applying a filter is not as slow as reading a file, but it takes several seconds to apply a filter.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Import</th>
<th>Export</th>
<th>Filter</th>
<th>Search IPv4</th>
<th>Search Vuln</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
</tr>
<tr>
<td>100</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
</tr>
<tr>
<td>1000</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
</tr>
<tr>
<td>10000</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Succeeded</td>
</tr>
</tbody>
</table>

MAC OS X results
The MAC OS X test results are presented in table 3.3. There was some bugs found, such as if a vulnerability is displayed at the moment of a search is initiated for a vulnerability, the search window will collapse in some cases. This made the Search Vuln test fail. When importing 1000 nodes the time from the user clicked import until the nodes were shown in the window were about 205 seconds. Since the 1000 nodes test were so slow, 10000 were never tried.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Import</th>
<th>Export</th>
<th>Filter</th>
<th>Search IPv4</th>
<th>Search Vuln</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Failed</td>
<td>Succeeded</td>
<td>Failed</td>
</tr>
<tr>
<td>100</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Failed</td>
<td>Succeeded</td>
<td>Failed</td>
</tr>
<tr>
<td>1000</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Failed</td>
<td>Succeeded</td>
<td>Failed</td>
</tr>
<tr>
<td>10000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Linux results
The Linux test results are found in table 3.4. The same problems as on MAC OS X using .gnu were found on Linux using the same version of .gnu. Importing and displaying 1000 nodes took about 85 seconds.

3.5.3 System test
The engine test section results showed that the implementation does not work well with the .gnu runtime. Therefore it was excluded from the system test, if the engine does not work as stand alone, there is no point in testing the engine with other modules. Therefore the system test were only done in Windows environment.

The test were performed by scanning two hosts with the Nessus scanner. The information were then extracted by the VAIH module for Nessus logs and converted into the
Table 3.4: Linux test results

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Import</th>
<th>Export</th>
<th>Filter</th>
<th>Search IPv4</th>
<th>Search Vuln</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Failed</td>
<td>Succeeded</td>
<td>Failed</td>
</tr>
<tr>
<td>100</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Failed</td>
<td>Succeeded</td>
<td>Failed</td>
</tr>
<tr>
<td>1000</td>
<td>Succeeded</td>
<td>Succeeded</td>
<td>Failed</td>
<td>Succeeded</td>
<td>Failed</td>
</tr>
<tr>
<td>10000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

VAIH IL format. The VAIH IL format were then read by the engine. There was also a self constructed VAIH IL format file containing additional (fiction) information about the two hosts scanned in Nessus, and a new third host. This new information was also read by the engine. The engine managed to merge the two files, which provided additional information to the Nessus scan (although it was fiction this time). The sequence of this test is shown in figure 3.5.

Figure 3.5: System test sequence

Because of time constraints, two modules could not be developed and is why fictional information was used.

During the test it was discovered that information extracted by the module that contained multiple lines contained wired ASCII symbols. The information were not lost, but it was found annoying. Due to time constraints there were no time for fixing this issue.
4 Discussion

This thesis shows that vulnerability assessment can be performed in many different ways, depending on the area and the specific needs. This thesis has shown different techniques that could be used to collect information about hosts through a network connection. Many of these techniques are implemented in tools that are used today. I think the thesis theoretical parts gives enough background information for persons who are involved in computer science, but maybe not in the security area, can understand the problems to solve. The theoretical parts also go into such deep that it might interest someone with a bit of knowledge in this area.

4.1 Implementation discussion

There were quite a few requirements of the implementation. The reason why more functional requirements were not taken into consideration were due to time constraints. More functionality, such as filters or search options, are always desired. I think my implementation constructed the ground work for this type of system, a developer with knowledge in C# shouldn’t have any problems with continuing the development of search and filter abilities.

The test results of the implementation (section 3.5) were a disappointment when it comes to Linux and MAC systems using .gnu. However, it is possible that some of the failed features will work on later versions of .gnu. For example, the filter did not work because the class used were not 100% implemented. Whenever the class is implemented it is a good chance that filter will work. But there were also issues with the slow performance and unstable GUI. These problems are more unlikely to go away any time soon.

On Windows the test result were more positive. But also the Windows implementation were slow when handling networks as big as 10000 hosts. The system test also showed that information extracted from a scan can contain ASCII symbols that were not intended, which is annoying.

4.2 Goals

The implementation discussion (section 4.2) concluded that not all optional design goals (section 3.2.3) were met. But the most important design goals where fulfilled when running Windows, which was the main design platform. However, the implementation becomes very slow when working with networks which contain as many as 10 000 hosts (details are found in section 3.5.2).

I showed how known vulnerabilities can be discovered. I also showed a method, by using the VAIH system, that improves the process of finding know vulnerabilities compared with just using one VA tool or by using several tools and parsing the information manually. Even if the VAIH system is not ready for the real world at this moment, it clearly shows how a system for gathering information can be designed, and parts of the system could be used in a real world implementation.

I think this thesis has shown how known security flaws can be discovered on remote hosts over a network connection, and how all the information can be handled. Therefore the main goal of this thesis is fulfilled.
4.3 Future work

One of the biggest problems is to define a wide Intermediate Language (IL) format. The IL presented in this thesis (Appendix B) is sufficient for the organization who requested the system. I have also found it sufficient when investigating other formats within other areas of computer security. I’ve also taken a look at a few tools that can be used for Vulnerability Assessment (VA) and found that these tools produce similar information. Future work may include building a better foundation for a IL format by deeper studies in this particular area.

The solution for the information handling problems does need another implementation in order to fulfill all the requirements. I still think that the design of the system architecture is a great solution for this problem, however, I would propose a web interface for the future. This would allow for cross platform support and would also enable easy use of databases. Databases can rapidly reduce the time for handling huge amounts of data, and are usually very stable.

Another feature that could be included is to link each found vulnerability to its online description. This feature would enable the user the easily access any new information produced about a specific vulnerability.
References


A Example VAIH module

This appendix contains source code for an example VAIH module. The example module is not meant to be the best possible module, it’s an example of how a module could be developed to extract information from a Nessus output file.

Listing 1: Example VAIH module

```perl
#!/usr/bin/perl

use XML::Simple;

$numArgs = $#ARGV + 1;

if ($numArgs != 2) {
    print "Usage: nessus.vahim <input_file> <output_file>
";
} else {
    $iFile = $ARGV[0];
    $oFile = $ARGV[1];

    $xml = new XML::Simple;

    # read infile
    print $file;
    $data = $xml->XMLin($iFile,forcearray => 1,keyattr => ['host'],keyattr => ['service']);

    #Open outfile
    die "Error: file_exist!*n" if -e $oFile;
    open(outfile,">$oFile") || die "Unable_to_create_file: "$oFile\n";
    binmode(outfile);

    #Print IL file header
    print outfile '<?xml version="1.0" encoding="ISO-8859-1"?>\n";
    print outfile '<file>'\n";

    my $nodeNr = 0;
    # Iterate through each node found
    foreach my $res (@{$data->{results}->{0}->{result}}) {
        print outfile "<t<node_id="$nodeNr">\n";
        print outfile "t<interface>\n";
        print outfile "t<addr_type="ipv4">$res->{host}->{0}->{ip}</addr>\n";
    }
```
# Iterate through each port found

```perl
foreach my $p (@{$data->{results}->{0}->{result}->{nodeNr}->{ports}->{0}->{port}}) {
    if ($p->{portid} != "") {
        print outfile "			<port number="\$p->{portid}">
        print outfile "				<service>$p->{service->[0]->{name}}</service>
        print outfile "				<vuln>
        print outfile "					<id type="Nessus">$p->{information->[0]->{id}->[0]->{id}}</id>
        print outfile "					<native-text>$p->{information->[0]->{data}->[0]->{native-text}}</native-text>
        print outfile "				</vuln>
        print outfile "			</port>
    }
}
print outfile "\t</interface>\n";
print(outfile); $nodeNr++;}
```
```
## Intermediate Language format

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<file>
  <node id=""
    <interface>
      <addr type=""/>
      <port number="">
        <service/>
        <vuln>
          <id type=""/>
          <src/>
          <native-text/>
          <verified/>
          <risk/>
          <comments/>
        </vuln>
        <comments/>
      </port>
      <function/>
    </interface>
    <nsinfo>
      <type/>
      <name/>
      <comments/>
    </nsinfo>
    <comments/>
  </node>
  <vuln>
    <id type=""/>
    <src/>
    <native-text/>
    <verified/>
    <risk/>
    <comments/>
  </vuln>
  <comments/>
</file>
```