Unifying Security Tools

A project Presented to The Academic
Faculty Of Eastern Washington University

In Partial Fulfillment Of the Requirements
for the Degree Masters Of Science in the
College of Science, Health and Engineering

David McCombs
June 1, 2011

Unifying Security Tools

The project of David McCombs has been approved and is acceptable in quality and format:

Approved by:

Dr. Carol Taylor, Chairperson
Computer Science
Eastern Washington University

Dr. Kosuke Imamura, Member
Computer Science
Eastern Washington University

Dr. Doris , Graduate Council Representative
Library
Eastern Washington University

Contents

Abstract

This project describes a proof of concept to unify various security tools and libraries. It focuses on basic functionality that comprises every network security assessment. The project is designed to not only unify tools by automating input and managing input but to smooth the learning curve of these tools for computer science students.

This report describes the current security landscape in regard to tools, libraries and methodologies. The design of the project is described as well as potential features in future work. The project is evaluated in terms of ease of use at both the programmer and security audit level and compared to manually running each step that was used in the initial audit script that was developed during the project. The scope of the project was limited to the basic audit steps without diving into details of the many substeps in each audit step.

Note: Words that are *italicized* are defined in the glossary.

Introduction

Security Overview

Since the Internet and World Wide Web has gained wider acceptance, security has been a growing problem. This problem has only increased as more companies, organizations and individuals have built and maintained their own networks which connect to the Internet. Further, lack of understanding of security principles have created well publicized incidents. A current example is retailer TJ Maxx, who lost thousands of its customer's credit card numbers after violating established practices.[5]

In early 2007, the retailer TJ Maxx discovered that the servers that stored several years of customer data, including credit cards were breached. The data was not properly secured and was compromised by attackers. Storing credit card numbers is in violation of standards created by banks that service credit card purchases. The end result was the loss of millions of credit card numbers, and banks started to offer millions in incentives for retailers to perform audits on their data collection and storage procedures.[3] The hackers used a variety of attacks to get this information such as installing *backdoors* and *Structured Query Language* (SQL) injection. The attackers were eventually caught and convicted, but much damage was done. This incident was entirely preventable.[10]

To combat these threats many types of security measures have been enacted, all of which are expensive in terms of both time and money, and as the TJ Maxx incident demonstrated, compliance is not always enforced. These measures include showing how to secure data, and what data should be kept,and implementing security tools to monitor security breaches and stop attacks such as *firewalls* and *intrusion detection systems*.

There are many existing security standards, all of which vary depending on what the business is and what types of data are stored. There are also exist-

ing standards that are more general. Standards such as those published by the Institute of Electrical and Electronics Engineers (IEEE) and International Organization for Standardization (ISO). These standards are comprehensive and cover areas such as data storage, encryption, and compliance testing procedures. However, the IEEE and ISO standards are not mandatory by any type of enforcement or regulatory agency and thus may not be followed. This lack of mandatory implementation exists for many of the industry-defined standards as well.

Governmental regulations such as the Sarbanes-Oxley Act (SOX), Health Insurance Portability and Accountability Act (HIPAA)[7], and Gramm-Leach-Bliley Act (GLBA)[11] have placed additional requirements on many types of companies and corporations. Oddly, these regulations place very few requirements on the producers of software and hardware that these organizations use. This lack of accountability on hardware and software manufacturers can open up these networks to vulnerabilities that can only be solved by placing security boundaries around the software. This can lead to a complex network that is difficult to manage. These requirements do not only address computing and network security, they add requirements to prove compliance, plus there exists other software and hardware standards that might be required in certain domains, such as developing and implementing hardware and software for the Department of Defense that will be used for classified purposes.

In addition to implementing security measures, these regulations require third party *security audits*, sometimes called *penetration testing*. These tests cover many aspects of network and computer security. While the extra government regulations dictate standards, which to an extent dictate the type of tests there are no standard tools or methods used by external or internal security auditors. There may be a set of standards within an organization, but between organizations and auditors the tools and methodologies may vary widely.

To complicate matters, computer manufacturers and suppliers of operating systems rarely configure the system in a secure manner "out of the box". This

leaves home users and small businesses who cannot afford to hire professional administrators lacking information on how to secure their computers and networks, leaving them increasingly vulnerable. The lack of secure configuration is compounded by an ever increasing number of software and system vulnerabilities.

In 1995, 171 vulnerabilities were cataloged by the Carnegie-Mellon University Software Engineering Response Team(CERT) and in the first three quarters of 2008 they collected a list containing 6058 vulnerabilities nearly as much as the 7236 vulnerabilities found in 2007. They have since stopped collecting this data. [2] In 2007, over 37,000 breaches of government and corporate computer systems were reported. [6] According to statistics listed by Lavasoft, an anti-spyware company, 90% of all home computers in the United States will get at least one infection, and according to Gartner in 2006 40% of companies will either get infected or targeted over the Internet by "financially-motivated" criminals.[9]

Current methods and tools used to secure computers have flaws but they do provide some benefit and are always improving. While firewalls and IDS are not the only tools used to combat attackers, they are the first line of defense. Both firewalls and IDS rely on having rules properly configured so malicious data can not pass through the firewall without alerting the IDS. The rules can be very complicated and are error prone, especially in a complex network that has several levels of traffic and users. It is a very complicated and expensive process, even for well-trained professionals. For home use, the environment is less complex, but so are the tools and level of protection.

Many firewalls built for home use take many of the decisions out of the customers hands, leaving them at the mercy of the default settings. On the other hand, few home users are educated enough to make informed firewall decisions. Consumer level intrusion detection is rare and what exists is mainly anti-malware scanners that are reactive. They are based on known malware signatures with little ability to detect new malware. The methods used to obtain access to these computer networks are also improving and often at a

faster rate.

Many of the tools used to defend and test networks are also used by attackers, but attackers can go further with brute force attacks that take time and skill. However, there are many attacks that are automated. These are used by many different attackers with differing motives, but are noted for being used by '*script kiddies*'. These attackers have little to no technical knowledge and completely rely on these scripts to carry out the attacks.

The weakest link in any network is the human element. Many companies routinely conduct security training for their employees to combat this threat. Despite this effort, *social engineering* threats remain a growing problem. These threats are made possible through weak passwords, downloading and installing suspect programs, opening suspect email attachments, and voluntary information disclosure. This is truly the most critical part of security. However, it is the general belief within the security community that no software or hardware solution will ever mitigate this threat, nor will any regulation.

Existing security tools either demand too much knowledge and user intervention or have too little interaction, both of which lead to lowered security. The "nag" alerts by verbose tools such as Windows-based firewalls annoy the user, leading them to turn it off. The tools that operate silently often only provide minimal functionality by default and make it difficult to configure for the "average" user. Security penetration tools are beyond the knowledge and skills of the "average" computer user, and even many system administrators. They require not only knowledge about the tool, but the systems and protocols that they are testing.[1] Regardless of whether or not security standards exist for an organization, the tools described in this paper are not specifically modified to meet them. This requires manual testing which can take longer, has a steeper learning curve, and is prone to errors and omissions.

More security information is available than ever before, and more news stories about security breaches have raised awareness, but that hasn't slowed down the number of security flaws and exploits. For home users, there may be no

good solution that will generally work other than improving awareness of scams and demanding operating system and software vendors to take security more seriously. For organizations and those interested in learning about these issues and how to avoid them the landscape is broad and cluttered with many tools and techniques making the learning and work environment complex.

Motivation

Computer science students learning security find a steep learning curve for many of the free and open source security tools and information about how to conduct a thorough audit is lacking. With each additional tool and protocol the learning curve becomes steeper. Further, these tools generally do not work together and often miss important information and suffer from false positives. That is, results from one tool have to be manually added to another and even after running a tool, much more testing is required to verify the results. For example, the results of a open port scan, which is usually very accurate, from *Nmap* may have to be manually entered into a vulnerability scanner or attack tool. This results in considerably more time required to finish the audit, and there is a significant chance that a certain target may be inadvertently left out.

In the professional world, these same issues occur, and usually on a larger scale, since a professional penetration tester may have potentially thousands of targets and dozens of protocols to account for in a single audit. Finding a way to minimize the complexity, while raising the quality of an audit will help security professionals to catch up to the attackers. Automated reporting is one way to help achieve this, since it will be able to report on everything found and when done manually this is another point where errors or omissions can happen.

Existing Solutions

There are integrated security tools and Linux distributions that come with security tools preinstalled. Some of these tools in the Linux distributions can be difficult to install and configure. However, the tools still lack higher levels of

integration. The integrated tools are very expensive ranging from a few hundred dollars up to tens of thousands of dollars. These tools are also not what the attackers use to probe for and exploit targets which means avenues of attacks and vulnerabilities may be missed by the auditor that will not be missed by an attacker.

There is some integration among various open source tools. The web-server vulnerability scanner called Nikto can integrate into Nessus and OpenVAS, which are generalized security scanners. Nessus and OpenVAS will integrate with Nmap, however it does not do well with a large number of targets. These tools do not integrate with many other tools which might logically be used to validate or exploit the vulnerability. Results from reconnaissance tools such as nmap are not integrated into the input of some vulnerability tools and output of various security tools such as intrusion detection systems and firewalls. For example, knowing immediately if an Nmap scan was detected rather than having to find out via searching through logs and email would greatly increase the value of the scans.

Threat assessment is the process of determining what assets to protect. Many current threat assessments often treat each node in a system as it is in a vacuum.[8] An arbitrary value is assigned based on how difficult it is to break in and how much damage they can do. This type of methodology often misses how a vulnerable but low value target may have elevated privileges to a higher value system or can gain privilege through more exploits leading to more access inside the local area network. This technique, called *chained exploits*, can be used as a base for further attacks into the network. Using multiple tools help find these multiple level exploits but can be difficult because of the lack of integration between them. There are also so many different possibilities to exploit a network, some are known, but many are unknown at any given time. This makes vulnerability scans that look for specific problems less valuable because older problems are invariably fixed, and there are always new vulnerabilities discovered and not reported.

There is an analog to this in the software testing field. It is commonly called *fuzzing*. Tools called fuzzers throw a mix of random and crafted input against a target. Care is taken to test all paths of execution so all states can be tested. These tools eliminate the need for a lot of manual testing and can be made more useful by adding techniques such as supervised and automated learning into the fuzzing process. This type of audit added with the more commonly used techniques add much better coverage to any security test but this area of study is still in its fetal stage.

Statement of Purpose

The purpose of this project is to determine whether or not integrating existing free tools and using the result to craft further input is useful as a technique for security evaluation. This will introduce an integrated software solution to assist in internal and external evaluations of networks. Proprietary tools that can cost thousands of dollars are not considered in the evaluation of this project. The cost is prohibitive and they are not used as much as the open source tools because of the fact that the attackers use them as well. Making a computer network 100% secure is an impossibility, but finding and fixing the majority of holes that the low to mid skill attackers can find will get a network much closer to that elusive goal.

The comparisons will be in the form of using results for this project to create a baseline for evaluation against using these tools independently. A very important factor in judging this project will be in evaluating the automatically generated report. Initially, the results will be more results-based, what was found and not necessarily the implications. Over time the project can be expanded to a knowledge base system that can determine the seriousness of what was discovered, such as chained attacks and network fuzzing.

Another possible metric for evaluating this project would be to measure how it works as an educational tool for computer science and networking students. The points of evaluation are based on usability and usefulness. It is hoped that

the program will assist networking students in learning the basics of a security audit while lessening the steep learning curve of the many tools and techniques used to perform these audits. A professional auditor will still have to understand the tools used in the same way that a programmer needs to understand libraries and how software interacts with the underlying hardware. The main benefit to the professional is the ability to easily use output of one application to craft input, whether automatically or by hand, into another program. This will save time and increase the accuracy of the scan.

If possible, finding professional auditors to evaluate the program would be helpful in evaluating the program. Time constraints and availablity of interested people will likely make this difficult. This project will utilize several disciplines in computer science; networking, security testing, and software design and engineering. The main focus is in contributing to security testing and education. Secondary concerns relate to the design of the program and the ease of extending its functionality in the future.

The initial state of the project will be a basic prototype to demonstrate how the included libraries are used to create a plug-in system and how custom audits can be written. While the entire API will not be written as the potential size and scope is nearly limitless. This will introduce the importance of scaling the library while keeping the learning and usability curve mild. However, a library of all the tools used in the prototype will be made available.

Each part of the prototype audit will use different methods to write the audit scripts. Object-oriented and functional approaches will be used, as well as a meta-programming example. These will be evaluated in terms of how well they will work as a general use method of writing custom audits and the steepness of the learning curve. An important consideration is how easy it is to write a custom audit. The programming skills of students can differ by a wide margin and professional auditors may not have any formal computer science education or may not even be able to program at all.

The evaluation that will be performed will consist of several parts:

1. Determine the ease of use and accuracy of the automated audit in comparison with running the individual tools separately.

2. Evaluate each programming approach in terms of flexibility.

3. Ease of interfacing with the libraries when writing custom audits.

Project Goals

This section lists the goals that determine the software choices and development. It should be noted that it is not the intent of this project to produce a complete working audit tool that covers the entire network security spectrum. The main goal is to produce a working prototype, called **NetProbe**, that can be used to focus and develop the program in a development team.

The main focus are legitimate security audits, where the auditor is authorized to test the network. It is not intended to be a tool used for nefarious purposes. However, It has always been the case that legitimate security tools are used by hackers. For example, locksmith's tools have both legal and illegal uses. There will be features added to the initial design so legitimate testers can make sure they are not attacking unauthorized targets.

It is also important to note that the initial plugin is targeting production systems. Thus, techniques that can bring down the system or corrupt data will be avoided. In later work, these techniques can be added to target mirrored systems that will not disrupt the business of the client if attacks do bring down the system. Because of this, the initial prototype will return important, but fairly shallow details. These details include the IP addresses off all machines in the network(s) and the types of network services running These details are typically obtained before the auditor gets into the attack and deeper enumeration and discovery tests. Therefore, the results will be incomplete but will still be useful for both academic and business world purposes.

Goal 1

The basics of the security audit will be implemented and tested. This includes several steps: enumeration, discovery, vulnerability testing, and reporting. Each of these steps can have many substeps which are ignored given the time and personnel constraints of the project.

Enumeration

Enumeration is the process in which IP addresses and associated names are found. For example, find all IP addresses associated with a URL, such as www.example.com or to take a range of IP addresses and try and put names to them, i.e. 192.168.0.0/27. The goal of this step is straight forward: find as many targets as possible without adding irrelevant targets. For example, after a search of all associated name related to www.example.com, all of the host names can be compared with the owners to make sure they are all owned by example.com. Of all the basic steps in a security evaluation, this one is the most consistant across all audit domains. It is also critical so that targets that the tester is not authorized to use are excluded, but all authorized targets are included.

Discovery

Finding the services running on each node, the operating systems used, and possibly existing *firewall* rules is the main focus of discovery. This includes testing ports for all network nodes to see which ports are open, closed or filtered. A port is filtered if no response is received from the target, this means the firewall is dropping the request or that no machine is connected to that address. An open port is where a non-reset response is given and a reset response identifies the port as closed. This presents a good picture of the firewall rules, but more involved testing is required to get more specific rules. The advanced firewall rule testing is irrelevant to the evaluation of NetProbe so won't be initially implemented. These kinds of tests send packets that are corrupted in some way or do not follow established protocols such as TCP.

Vulnerability Testing

Vulnerability testing can involve several steps. One is to actively probe for vulnerabilities that exist on the machine being tested. Another possible step is to try and break in to nodes in the network. Many of these techniques, such as fuzzing, exploiting *buffer overflows*, and *SQL injection* that attempts to write to the database can damage the target and will not be initially included. A more advanced technique is to discover *chained exploits*. An example is to break into a FTP server to try and get access to private networks behind the server.

The one step in vulnerability testing that will definitely be implemented is using the network service data found in discovery to search online vulnerability databases for known problems. These discoveries will not denote problems that do exist in the target, but will alert the tester to potential issues. The other techniques such as running *Nikto*, *Nessus* or *OpenVAS* will be added if time permits. While these additional tests will increase the usefulness of a scan, they will not add much to the value of the initial evaluations of the project.

Reporting

The last step is to report what was found. This involves taking all the collected data and generating a readable report that can be acted on. Initially, the report will cover what was found, but have little extra information about the implications of what was found. For example, if the tester successfully attacks a target and installs malware, certain types of port scans can help evaulate if the firewall will get in the way of it communicating outside the network. In future work a knowlege base can be developed to report these kinds of critical implications of what was found but is not all that relevant to meeting the goals of the initial evaluation.

These steps will be run and the final report will be used in the functional evaluation to determine the value and potential of these automated scans over running them manually.

Goal 2

There is nearly an infinite number of variations in an audit scan that are determined by goals of the scan, type of the target and any regulatory requirements that must be met. The difference in an audit of a bank's systems and the scan a student performs in the security lab should be obvious. For that reason a plug-in system that is both simple and powerful is an important goal. The ability to add in new functionality into any script at runtime should be included in long term plans. This will enable the auditor to view, evaluate and respond to unexpected or interesting results during a scan. There needs to be a reasonable tradeoff between lines of code in the audit script and its flexibility.

To evaluate the possibilities, the NetProbe audit plugin will try various types of software paradigms. These will be evaluated by several criteria. It should be easy to learn. Not everyone who might use this program has programming skills, but most testers should be able to write a script that is nothing more than making a list of audit steps and tools to run and the underlying libraries can perform those steps and properly interact with other steps as needed. It must be flexible enough to be useful at several levels of audit detail and be able to handle customized code that can be injected into the scan at one or more points.

These two criteria are in conflict and tradeoffs will need to be made. For example, a short list of commands could be developed that require no special programming ability, however, it would be hard to be flexible enough to be able to write scripts for simple and advanced audits. On the other extreme, having a set number of libraries for basic functionality and then forcing the audit script writer to come up with the rest of the functionality with little library support would make the system useless as the time required and knowlege requirements would be excessive.

Design Considerations

The following sections describe the considerations given to the design of the software and how it relates to the evaluation of the project.

Languages and Frameworks

The language chosen will obviously have the biggest impact on the choices for frameworks and interfaces. It also has a major impact on the ease of development and its ease of use as a security tool. The basic requirements are that the language is free, easily available, reasonably easy to learn and use, and is simultaneously cross-platform and able to effectively leverage the underlying operating system and hardware. To keep the options down to a reasonable number, only those languages that are used in EWU's computer science department, in one or more courses were considered. While this constraint removes excellent languages such as Lisp and Haskell, it will help ensure that anyone who picks up the project for a class or Masters project can do so with little effort. It also ensures that the language usage is widespread enough to be used by a large portion of the intended audience. The languages to be evaluated are C, C++, C#, Ruby, Java, JRuby, and Python.

The issue of cross-platform is important but not all tools that can or will be used adhere to this requirement. Many of the basic tools used are cross-platform, but more advanced tools generally are not. Sill, it is important that the basic audit framework be useful for auditors regardless of their platform of choice.

Another important consideration is *domain specific language* support (DSL). The ability to write simpler languages with a given language and dynamically adding features to the scans is one that will increase the flexibility and value of any audit. Features that help support this include *dynamic typing*, *open classes*, inheritance, lambdas(*anonymous functions*), and *closures*.

C/C++ - These two languages are compiled to machine language so are

generally faster then the other languages evaluated. The bottleneck in the project is running applications that can take hours to complete. Because of this execution speed is not important. The only other advantage these languages hold is that it allows for direct access to system libraries and hardware such as network cards when needed. However, the other languages do have bindings to these low-level routines, and often in platform agnostic ways. With care, these languages are cross-platform with the added requirement of compiling the code for each platform. A big disadvantage is that these two languages are very verbose, with C++ being the most verbose by far. They are also not flexible in terms of easily writing *domain specific languages*.

C# - This statically-typed object-oriented language from Microsoft has little to recommend for this project other than it has some support for *anonymous functions* and *closures*. The syntax is familiar to programmers that know Java and C++. C# is not cross-platform in the sense that the only supported runtime(.NET) is Windows only. Linux support is achieved via the mono project. Mono is not a Microsoft supported project and as such it is always behind the .NET development curve which means there may be incompatibilities when moving across platforms.

Ruby – A dynamically and strongly typed object-oriented language with strong meta and functional programming support. Its advantages are flexible frameworks and it is easy to learn. While the basic syntax is easy to learn it has commonly-used features that may not be intuitive to programmers familiar with C or Java, such as *closures* and lambdas. As such it can be described as easy to learn but difficult to master. It is also commonly used for system administration tasks so it can run external scripts and programs easily. There is also good support for linking to C code via built in libraries and external libraries such as Ruby-FFI. Ruby is well-known as a language of choice for DSL writers. The Ruby interpreter is relatively slow and its C support is so good that much of the Ruby standard library is written in C. Its downsides are the execution speed, which is a non-issue for this project and lack of *native*

threads, but does have *green threads*. The threading is done in software, so taking advantage of multiple cores is impossible. The recently released Ruby 1.9 has hardware threading support, but much of the standard library is not yet thread-safe so native threads in Ruby 1.9 are intentionally hobbled.

Java- A statically-typed object-oriented language with solid libraries. It boasts good execution speed because of the *just in time* (JIT) compilers and has outstanding threading support. The syntax is familiar to CS students at EWU as well as those familiar with C++. It is also quite verbose. and has little in the way of acessible DSL support and does not integrate with C and the underlying operating system as well as other high level languages. It is possible to access native libraries using the Java Native Interface (JNI) which is fairly complex.

JRuby – This is not a language per se, and is not used in the department. JRuby is an implementation of Ruby written in Java. This means that JRuby has the dynamic-typing and meta-programming support of Ruby with the speed and thread support of Java. It also has access to the Ruby and Java libraries as well as supporting access to C and C++ libraries via JNI or FFI. The downsides are that many of the ruby libraries that are written in C are not supported. These libraries were rewritten in Ruby, but still lack some of the low-level system access of Ruby.

Python – A language similar to Ruby in scope, but with a different syntax. It is also does not have nearly as good support for dynamic programming and has limited closure and lamba support. Otherwise, Python has similar advantages of Ruby. There is Java and .NET support for Python, such as Jython and IronPython, but these are not considered simply to keep the number of options down. Also Jython development has lagged in the last few years.

Frameworks are libraries that simplify tasks such as database development and dynamic web page generation. The frameworks that may be used depending on language and interface choices are:

.NET – Supports a wide range of uses from desktop to the web for C#, also

has support for Ruby and Python. However, Microsoft has pulled Ruby support and the ,NET Ruby project (IronRuby) has only one part-time volunteer developer.

Ruby on Rails – A powerful web-based framework. It is modular so can be used in other contexts. Has a simple and powerful library to manage databases called Active Record and Nokogiri, a terrific HTML/XML parser and builder.

Django – A web-based framework for Python. Similar to Rails in scope and modularity.

Active Objects – An object relational mapper for databases written in Java and modeled after Active Record

Java Server Pages(JSP) and Servlets – A dynamic web page and server library for Java. There are also many database access libraries available for Java such as Hibernate.

Interface

Interface design is not a large part of the initial prototype design and evaluation. However, ease of use in terms of actually using it and writing code to leverage the interface is considered. In the software world there is, in general, three types of user interfaces. These types are command line, graphical and web.

A command line interface is the most basic, and despite that it is very powerful. In fact, most of the programs that the project uses are command line and any program that will be useful to this project will support the command line. This makes integrating them into the project, regardless of the interface used much simpler. Providing a command line only application would not satisfy the goal of simplifying the process of the security audit, and will not be used in the prototype. It might also complicate the goal of platform independence by relying on the underlying operating system. If command line was implemented, writing a custom prompt inside a graphical interface would keep the platform independent goals intact at the cost of adding more complexity to the UI interface work. Adding an option to use it, or provide a command prompt at certain

points of the audit may prove useful in future work.

A graphical interface is the next option. While it does add complexity in terms of development, the ease of use advantages are a good trade off. The main issues are using an appropriate GUI library to maintain platform independence, using a library that doesn't have a steep learning curve, and abstracting writing to the interface so that the library used is irrelevant. Solving the first issue depends on the language used, although there are cross-platform libraries that support multiple languages such as QT and wxWidgets. There are also cross-platform libraries for single languages and runtime environments such as Swing or AWT for Java. A command line and GUI can be implemented over the same code with a little foresight and planning.

The last choice is a web interface. It has the advantages of ease of use, is fairly easy to develop, and is trivial to run on multiple platforms. For this project a disadvantage is that many tools used take a lot of time and it is difficult to transmit that information in a web page without using Asynchronous Java and XML (AJAX). AJAX, while improving all the time can be error-prone and cause a lot of exceptions. This issue is not serious, but adds more complexity for little gain over a traditional desktop graphical interface.

Parts of the project may greatly benefit from multi-processing, which may be more difficult in a web architecture depending on the frameworks and server chosen. Another advantage is the client/server model. This helps force clean design and allows the most computationally intensive tasks to be run on more powerful machines. A big advantage is that the user interface can be trivially separated physically from the back-end. This gives the ability to run all the scans remotely, which would make it easier for students to use. An example of this is to keep a server running in the security lab at EWU, and students can connect to it and run the scripts at home via a web browser, with no additional software required. This decoupling is possible for graphical and command line interfaces but does require installing software on the client machine. Good design can be accomplished with any interface type with usage of patterns such

as Model-View-Controller(MVC) but may not be as strongly enforced outside of web frameworks.

Initial Prototype Choices

The choices made for the prototype are as follows:

Language

In order to allow maximum flexibility in terms of software design, as well as familiarity and access to the greatest number of libraries, JRuby will be used to develop the prototype and evaluate the software design options. This will allow easy integration of various network security tools and libraries while maintaining access to a solid runtime environment.

Interface

The interface type will be a graphical user interface using *Swing*. There are other options within JRuby including SWT, Tk,and Jambi, the Java bindings for QT. Swing is taught at EWU so is more easily accessible. Perhaps the biggest negative aspect of Swing are its layout managers that are used to place GUI components on the screen. They are complex and very verbose. To overcome this a third party layout manager called mig layout will be used. This will ensure that the effort needed to present a user interface is minimized so more time can be spent on developing and evaluating the more interesting and relevant parts of the project.

Frameworks

With the above choices, the potential frameworks have been minimized. Since the data collected with be stored in a simple database, Active Record will be used. This will minimize the amount of code and time spent in reading and writing to the database as well as managing relations between the tables. Other Rails tools such as rake tasks, console access and database migrations will be

used. Almost any database can be used with ActiveRecord. It is trivial to switch to a different database at any time without code or schema changes. A simple change in a configuration file is all that is needed. For development purposes MySQL will be utilized.

Software Design

The design of NetProbe will follow the MVC model which keeps GUI, database access, and business logic separate. Initially it will feature a simple audit plug-in system that will dynamically run various audits. The audit plug-in will feature a different software design approach for each major step that can be evaluated to see how easy writing a plug-in will be. These approaches include various object-oriented designs, meta-programming and domain specific languages.

This mix of approaches will initially result in more libraries being needed to be written, but then when they are evaluated and reimplemented, the excess libraries can either be modified or thrown out. This is known as prototyping. It will also mean that the audit plug-in script will not look consistent. Since each portion of the scan will have different requirements, parts of the audit that are the same across all of the audit steps will be abstracted away so these parts will not impact design. A more detailed explanation and examples will be given later in the specification section.

Tool and Library Selection

For the initial audit plug-in, the following tools will be used. It is a small list but will provide enough functionality to cover all the steps in a typical audit. There are many other tools and techniques that can be used to give more detailed results. A few of these will be documented in the future work section.

Host – A program that performs Domain Name System (DNS) lookups. This is used to map IP addresses to host names and vice versa.

Whois – A utility used to lookup information about domain names such as who owns it,

Nmap – An all purpose port scanner that can be used to determine if network nodes exist, what ports are open, and often what services, version and underlying operating systems are at each port. Results are often complex, but there is a ruby based parser to handle running and parsing Nmap during a scan.

Open Source Vulnerability Database (OSVDB) – This is a web-based vulnerability database that maintains lists of known vulnerabilities in open source and proprietary software. It provides a simple interface to retrieve requests and also has the option of downloading the entire database. The prototype will manually retrieve vulnerability information instead of downloading the entire data set. There is a 100 request limit per day when using the API, but is simpler in the long run. This query limit can be increased by contacting the OSVDB managers. The online database is updated daily, and is a large download.

Nikto - A vulnerability scanner used to test web sites.

There are other libraries that may be used in development to assist in using the above tools.

Nmap-Parser – This Ruby library is used to parse the information received by nmap scans.

Whois – A simple ruby gem to run who is requests.

Jpcap – If direct raw packet access is required this Java library that wrap the C library libpcap will be used.

Open3 – A Ruby library used to manually run other programs. This library manages input, output and error messages automatically so is more useful that the built-in methods to run external programs.

If a program or library does not have an existing library to run and parse the results, custom libraries will be written. Also, other libraries that handle tasks such as network address calculation will be written using existing ruby network libraries.

Prototype Description

The prototype, as mentioned is a desktop graphical application written in JRuby, utilizing the Swing library. The only goal for the graphical interface is to make sure it is straightforward to use.

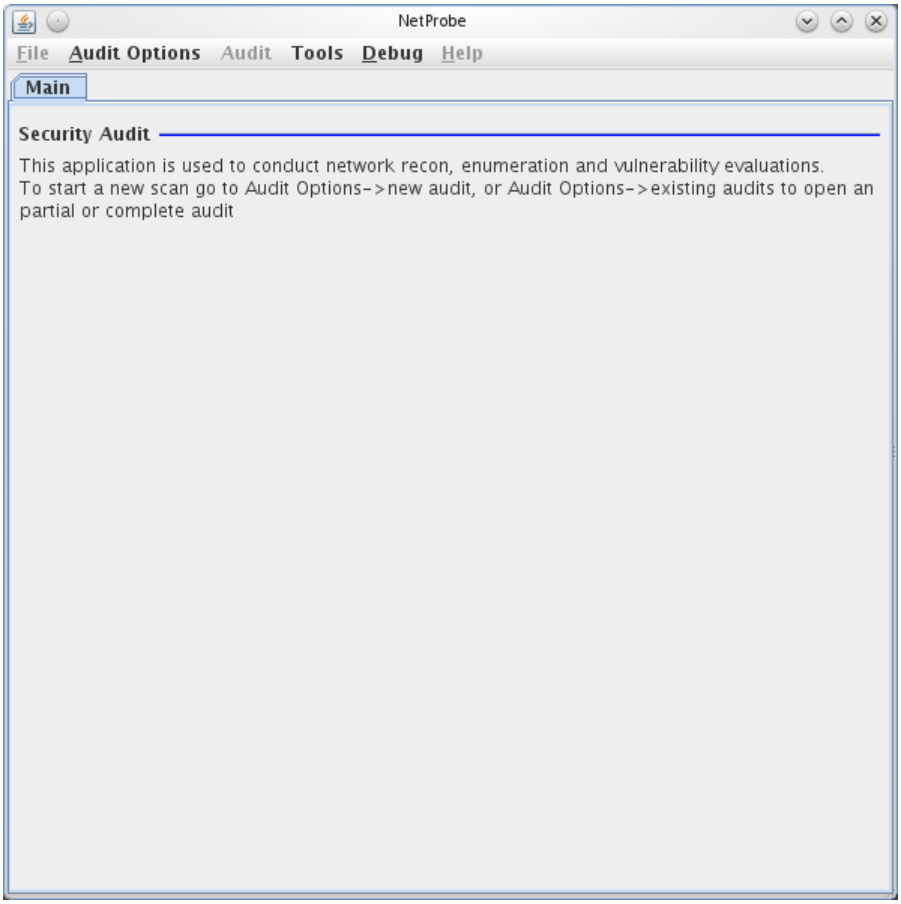

Figure 1: Swing-Based Interface

Other than Swing, all other code written is done in Ruby, including the Swing listeners that control the actions after a button or menu item has been clicked. The Swing code is packaged up in a Java Archive (JAR) file, all other code is stored in text files. It is possible to compile the Ruby code into Java .class files and then archived into JAR file, and this is a likely method of distribution, but adds nothing to the project at this time. This method also allows for the

distrubution of a single JRuby JAR file which can be run by any current Java Virtual Machine(JVM).

The basic layout of the application follows the Model View Controller (MVC) pattern (*Figure 2*). The application is split into logical portions. The view holds all the code used to generate the interface. The model performs routines on the database and the controller ties them together. The controller consists of listeners, and the plug-in system. All of these parts are in the top level folder called app. In addition to the MVC structures there are libraries discussed below.

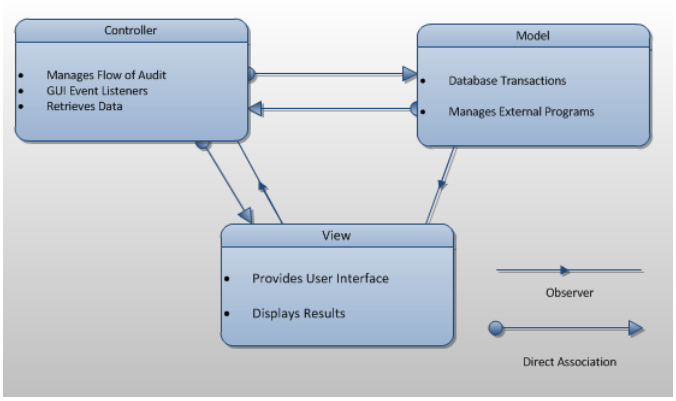

Figure 2: Model View Controller

The last portion of the application structures are helpers. These are similar to libraries, but are generally simpler and assist with parts of the application. There are helpers that assist with setting up and storing information in the database. This could have been done in the model classes, but was done this way to keep the model class code clean and not break the ActiveRecord convention that one model class represents a table and that a model instance is a row in the database that Active Record tries to enforce.

A good method to write messages to the interface dynamically is not part of the initial prototype evaluation. As such, it is mixed into the code to run the various scans at not only the plug-in level but the application level. To avoid unreasonable tight-coupling, any UI code in the application libraries are abstracted and in some cases dynamically passed in as lambdas, This will make is simple to pull out the UI code after evaluations are completed, but it does increase the size of the plug-ins somewhat.

Libraries

There are two different library structures. One is in the app directory, and the other is a top level directory. The application libraries contain functionality to perform tasks that are application specific and may know about the database structure and make model calls. An example is a custom audit plug-in library to talk to the external library that handles Nmap functionality. The other libraries are standalone and know nothing about the application. For example, scripts that run external programs such as host or nmap or calculate IP address ranges.

Plugins

The plug-in system is simple. The plug-in file(s) are stored under the controller directory called process to denote custom audit processes. The plug-in controller can dynamically start and manage the plug-in. The code is simple, as shown below.

```
1  class AuditRunner
2    def initialize target,view,noise, audit_process
3      require 'app/controllers/process/'+audit_process.name+'/'+
           audit_process.name
4      @class_name = audit_process.name.gsub!(/^[a-z]|\s+[a-z]/) { |a|
           a.upcase }
5      @process = Object.const_get(@class_name).new target, view,
           noise, audit_process
6    end
7
8    def run
9      methods = Object.const_get(@class_name).method_names
10     methods.each do |method|
11           res = @process.send(method)
12           break unless res
13       end
14   end
15 end
```

Listing 1: Audit Plugin Runner

The constructor for AuditRunner automatically calls initialize which sets up the state of the object. There are four arguments, target, view, noise, and audit_process. *View* and *noise* are not yet implemented in an audit plug-in. They are meant to allow for options to adjust scans to try and be stealthy or not, and to trigger specific functionality that would only relate to specific types

of scans such as external and internal audits, and also target wireless networks. *Target* is the target the user entered into the audit form. This can be either a URL or IP address. The argument *audit_process* is retrieved from a drop-down list that dynamically queries what plug-ins are available and is an Active Record model object. This is to minimize the chance that an invalid plug-in is selected. Inside the initialize method, the proper plug-in is imported into the runtime. *Require* is similar to #include in C, but a large difference is that it is not a keyword, it is a method and as such can be used any place in the program. The next line simply ensures the beginning of each word in the name is capitalized. This is necessary because in Ruby a class name is a constant and constants are denoted by the name beginning with a capital letter. The last line in *initialize* takes the class name as a string and calls the constructor as if it were a class. In Ruby, variables that begin with the sigil @ denote instance variables, variables with no special character are local variables.

The method called *run*, controls the flow of the program. The only method required by a plug-in is *method_names* which must return an array of method names as strings in order of being called. The run method then loops over the array, dynamically calling each method. The loop terminates early if a false value is returned by the plug-in method currently running. Any error messages are generated by the method that returns false or can be generated in the actual library that caused the audit to fail.

The prototype methods get fairly complex depending on the paradigm used and will be described later.

Utilities

As the main purpose of this application is to conduct security audits, much of the work completed for evaluation is in this area. However, there are a few items not directly related to those goals that were implemented or stubbed out and left as future work. This additional functionality can be used to augment audits, or be used on its own. These extras are located in the menu bar under

tools. The first one is simple, it simply checks to see if the application can connect to the database. The second item is a simple IP address calculator. It can calculate IP address ranges from given IP addresses, subnet, or *CIDR*. For example the last non-grayed item simply starts up the packet sniffer Wireshark. More about the packet sniffer will be discussed in future work. The rest of the items are grayed out. These include SQL Injector, metasploit, audit rule generator, HTTP fuzzer, and Google hacking. These will be covered in future work.

Database Structure

The structure of the database tables is kept as simple as possible. The full diagram of the structure is located at the end of this section. Some fields in tables are not yet implemented because they represent unimplemented features. The "base" table is Scan, and all other tables relate to it in some way. The pertinent information stored here is the target name. There are five other tables that relate to Scan in a one-to one or one-to-many relationship. The only two that directly relates to the rest of the scan are Iprange and Node. The other three directly related, are either unimplemented or for statistics.

Iprange stores the IP address ranges as a high and low value. This assumes a continuous IP range, but if the range is disjoint, more than one entry can be used. Each IP address in the range(s) also maps to a Node. A node represents a potential machine on the target network that matches the IP address. It could represent multiple machines in a private network, or a cluster controlled by a proxy or load balancer, these more complex situations are discussed later. The Node has two tables, both with a one-to-many relationship. Hostinfo stores whatever DNS information was found for this node. Port represents a port on that specific Node. Depending on the audit, all of the possible states of the port may be represented. These states include open, closed, and filtered among others.

Port has two tables. One is Ostype, which holds one or more probable

operating systems that the node is running. The reason Ostype relates to Port and not Node, is to help detect routers that use port-forwarding. The other is service, with has a one-to-one relationship with Port. This stores information on the application that is running on an open port.

The last general table is Vulnerability, which lists probable vulnerabilities for a given service. The prototype audit only checks for vulnerabilities with services, so there currently is no relationship with the Vulnerability table. Vulnerability will also eventually relate to Ostype.

The other two main tables are program specific, Nikto. Which holds Nikto scan data for specific ports and is related to Node. When further developing support for more third party security tools, tables must be included for them.

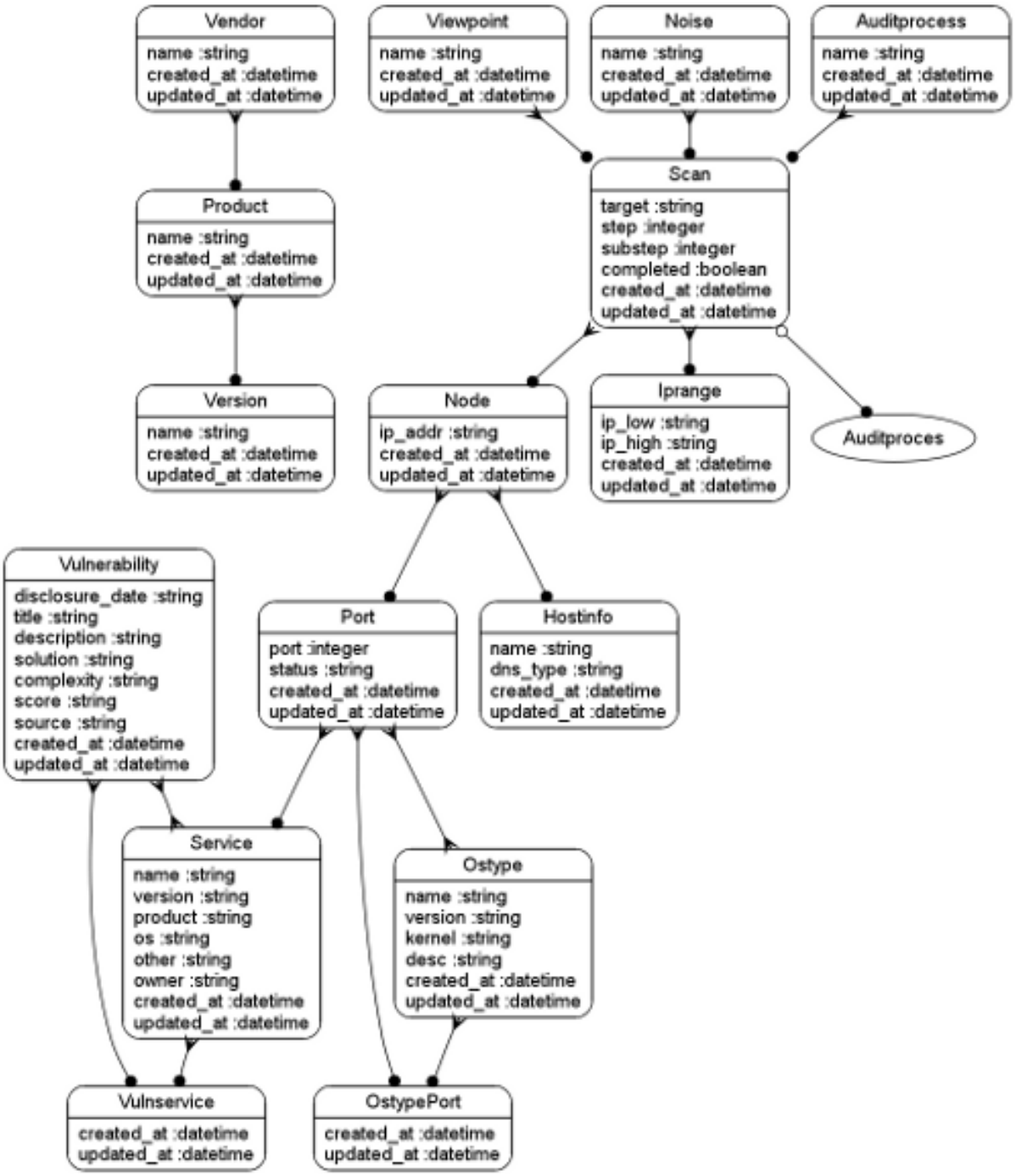

Figure 3: Database Model

Prototype Evaluation

This section describes the two ways that NetProbe will be evaluated for this Masters project: the audit plug-in initial design and the results of testing it against a target with both single and multiple addresses, where the multiple address target may or may not have continuous IP addresses. This should provide a wide enough scope to determine not only the value of the prototype plug-in, but should prove useful in further development. The same target will be audited manually using the same tools.

Results between the audit plug-in and manual operation will be compared. The most relevant areas of evaluation are the difference in the number of targets found, and ease of use. Since the same tools are used for both the manual and automatic audit, the accuracy of the tools used will not be different, given the same input. The evaluation is therefore focused on the input for these tools that is produced automatically and manually.

Prototype Audit Plugin

The initial plugin created contains four different parts that directly relate to the basic steps of an audit. These steps as described previously are: network enumeration, discovery, vulnerability testing, and reporting. The implementation of each step is detailed below.

Plugin Setup

The plug-in is a single class, with large methods that control a single-step. The instance variables hold references to the writer classes developed to write to the user interface and the various panels. The writer classes help decouple the plug-in from the different areas of the user interface. This will help make it much simpler to radically change the user interface, including changing graphical libraries. This approach also supports radically changing the interface at runtime to allow for customized interfaces for each audit step if necessary.

Ruby supports *open classes*, including core classes and modules (which are special cases of classes). The Ruby object model looks complex but it allows for a lot of flexibility because all classes can be changed at runtime. All classes are instances of Class, and the superclass of Class is Module, and the top-level class is called Object which is also an instance of Class.

What this circular model says is that all objects in Ruby share not only a common base class but also a module. There is a module named Kernel that holds common functionality that does not belong in an object base class. Methods such as require, puts, exec, and fork are contained inside Kernel. By being able to open Kernel, many types of functionality can be given to all classes within a runtime, most of which look like new keywords. It can also be used to add globals that are not global in the typical usage of the term. Visibility for methods and other object members can be changed during runtime, include Ruby API classes. An example is including the Singleton module into a class. Doing this sets many methods to private including the constructor and adding methods related to getting a reference to the object.

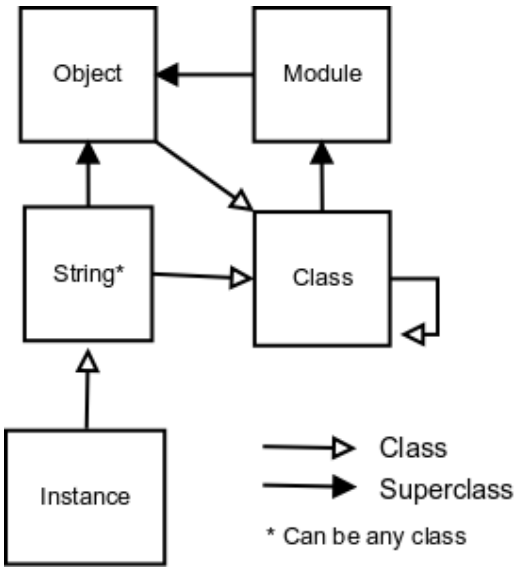

Figure 4: Ruby Object Model

Ruby also supports *closures* in several ways: blocks, procs, and lambdas. Blocks are essentially inline *anonymous functions*, and procs and lambdas are both blocks that can have references to them. There are many differences be-

tween blocks and lambdas, but essentially, blocks don't enforce having the correct number of arguments passed in a call, and lambdas simply just return if the return keyword is invoked. In procs if return is used, it returns from the point the proc was defined. This means that returning in procs is not always safe because the return pointer may be pointing to a stack frame that was already popped. However, all three types are closures because all variables referenced inside the block,proc or lambda are bound to it and these bindings remain through the entire life of the closure, no matter where they were declared or passed.

The only oddity at the class level of the plug-in leverages the dynamic and open nature of Ruby, as well as *closures*. The one piece of information that might be needed anywhere in the plug-in and as well as application libraries is the reference to the Scan model class. From this model, any part of the program that is allowed to know the database structure can traverse through the models to any point it needs to. In static languages such as C this would likely be made available with globals with no control access or wrapped in a struct and its value passed around as needed. In Java, it would either be stored in a static method or the reference passed around. The problem with passing the value of a struct or object around is that many objects and methods would have to accept this as an argument, even if it might not always need to use it. This makes method signatures larger, and tightly couples the models to many different classes and modules, making the libraries harder to write and maintain.

Ruby has several methods of manipulating scopes, and one of these are used to make the reference to Scan a controlled global variable. This is a nested lexical scope and is sometimes called a flat scope.

```
1  lambda {
2    scan=nil
3    Kernel.send :define_method, :scan_ref= do |ref|
4      scan = ref
5    end
6    Kernel.send :define_method, :scan_ref do
7      scan
8    end
9  }.call
```

Listing 2: Controlled 'Global' Variable

The lambda is created and instantly evaluated. Inside the block two methods are created dynamically that closes over the reference scan. which is now scoped in the two new Kernel methods and is accessible nowhere else. This makes the scan reference available everywhere in the application once the lambda is run and the reference set. The only issue in using this method is making sure that scan is not changed by another script running in parallel. The above code does not handle this currently because there are important implications to deal with when deciding whether or not to allow concurrent scans. This decision should wait until one of the approaches being tested is decided upon.

The send method allows for sending messages to any available method, this is called dynamic dispatch. This allows for two techniques: deciding at runtime what method to run and it gives the ability to call private methods. In both cases a message is sent to the private Kernel method define_method, which is one of the built in methods to dynamically create methods. scan_ref, and scan_ref= are the two method names, and in the case of scan_ref= the block argument is the argument for the newly created method.

An interesting property of Ruby is that mutator methods that end in '=' can be called in various ways. Such as: scan_ref= var and scan_ref = var. This allows for a more natural syntax.

Other information is collected and acted upon when the audit initializes. A reference to each GuiWriter used is initialized and various accessors and mutators are dynamically created. GUIWriter is implemented in Java and is in the GUI JAR file. GUIWriter can connect to any swing component that supports being written to. Examples include JTable and JTextArea. This allows maximum flexibility in creating user interfaces. This code exists in the Audit class, which is the base class for any plugin. The following code shown below creates all the dynamic methods along with an example: creating an append writer.

```
def self.set_writers writers
   writers.each_pair do |name, ref|
     create_accessor name, ref
     create_writer name, ref,false
     create_writer "#{name}_append".to_sym, ref,true
     create_clear name, ref
   end
 end

 def self.create_writer name, ref,append
   define_method name do |str|
     if append
       ref.appendText str
     else
     ref.writeText str
     end
   end
 end
```

Listing 3: Setting up GUI accessors and mutators

This code takes a hash with the name of the writer as the key, and the value is the reference to the writer object. The code then creates accessors, writers, and methods to clear all text from the GUI widget. Accessors, like the name implies returns a reference to the GUI writer. The writer methods automatically take care of common writing functionality such as appending, newlines, and moving the carat to overwrite from the top of the text box.

This approach allows for the script writer to have full control of the interface as there are no static, predefined method names of writing to the GUI within the plugin libraries. If a custom panel only has one place to write to(the default has two), it is created here. If the plugin writer needs more, he may create and name them at will and they will be accessible. Note that there is still just the GUIWriter available it is the reference name to the functionality that is being created via dynamic wrapper methods.

Besides, the flexibility this brings it makes the references "global" in the plugin object without having to maintain object variables or pass them between methods within children of Audit.

The hash is created when set_writers is called:

```
Audit.set_writers :message=>@panel.getMessageArea,:results=>@panel
    .getResultsArea
```

Listing 4: Setting up GUI accessors and mutators

The target is also validated and stored if it is in fact a valid IP or IP range, or valid domain name format. When everything previously listed is initialized the first audit step is called. Each interface component that can be written during runtime is required to implement getMessageArea() and getResultsArea() to properly access the current audit interface panel. How this method creates flexible interfaces is simple. If a panel on the interface has one or more than the current the writable widgets, the only code in the audit script that needs to be changed is the hash created in the set_writers() call listed above. This also allows a team working on this project to split the work between audit and library writing and interface design. All that the audit writers need to know from the interface team is the name of the methods to access the writers and the purpose of each writer.

Enumeration

The code for network enumeration is a fairly straightforward imperative-like programming task with a class in the application library for enumeration performing the bulk of the work including writing messages to the interface and saving the results to the database. What is written to the interface however, is determined by the plug-in writer. This is useful to plug-in developers that want different levels of reporting, instead of being forced to use hard-coded messages.

The plug-in has references to the interface writers and several helper classes. The relevant model classes are used to store the data that is instantiated for the enumeration libraries to use. It also needs to get the Scan reference.

As a convenience, a class that dynamically manages the enumeration process was created called Runner. This class takes an array of strings with the name of the enumeration steps that need to run on the target, as well as executing the interface writer procs, that are passed to this class as a hashtable. The string correlates exactly to a method name. This allows for maximum flexibility

in being able to easily write dynamic access to various levels of detail in the enumeration scan. The only methods in the enumeration library thus far are all used in this scan, and covers the most basic needs. These methods are *ip_range, reverse_dns, name_dns, add, confirm_nodes, ip_dns, and save!*.

- ip_range: Calculates the range of the IP addresses when giving a *CIDR*.
- reverse_dns: Attempts to match an IP address with one or more domain names.
- name_dns: Performs a DNS search for the given domain name
- add: Adds the data to the Scan model wrapper.
- confim_nodes: Displays the popup to confirm all nodes found in the test and to eliminate invalid nodes.
- ip_dns: Performs a reverse DNS search for MX, and DNS records.
- save!: Takes the results stored in the wrapper class in the appropriate models and saves it to the database.

The enumeration code written for the plugin contains two parts. Code to report progress by writing messages to the interface and the code to run the enumeration scan. The interface code is implemented as a hash of Procs which is called by the enumeration application libraries.

```
1  procs = {}
2  ip_range_msg = {}
3  ip_range_msg[:begin] = Proc.new {results_append "IP address ranges
       :"}
4  ip_range_msg[:end] = Proc.new {results_append "Added all ip
       address ranges"}
5  procs[:ip_range] = ip_range_msg
6  reverse_dns_msg={}
7  reverse_dns_msg[:begin] = Proc.new do
8    results_append "Performing reverse DNS"
9    results_append "A popup will appear to confirm domain names"
10   message_append  "Performing reverse DNS via nmap list scan"
11 end
12 reverse_dns_msg[:end]= Proc.new { results_append "Reverse DNS
       completed"}
13 procs[:reverse_dns]=reverse_dns_msg
```

Listing 5: Interface messaging

The key for each hash element denotes where and when to execute the proc. For example the ip_range key points to a hash with a begin and end key that are executed during the ip_range evaluation. This method allows for more detailed

feedback during runtime. Otherwise this has to be done in the audit plugin itself which would mean less feedback, or to get the same level of feedback the audit plugin writer would have to explicitly run each substep within an audit step.

This code only prints to the results box which is the bottom text box in the interface. To use this method to also print to the result text box, which is the right-hand side of the interface, adds a lot of complexity. To avoid this a reference to the result text box is simply passed into the Runner constructor and saved as a Class instance variable. Initially all writing to the interface was done without procs and there may still be some writing to the results text box in the original way that was left in the code. This is not an issue until the interface writing method is evaluated as the ideal solution. It may even be the ideal solution bcause it offers detail feedback as well as a robust custom system.

The code to run the enumeration scan is much simpler with all the heavy work done by a method in Runner called *run*. The constructor to this class takes in a list of features requested and the target(s).

```
1 run=Enumeration_Runner::Runner.new(@enum_features, target_data,
      @scan, get_message, get_results)
2
3 begin
4   run.run procs
5 rescue => e
6  Java::views.main.popups.Message.showConfirm e.inspect,"problem"
7  return false
8 end
```

Listing 6: Enumeration Audit Script

The Runner object is instantiated in the first line. The first argument is a hardcoded list of symbols of the feature name. In this case: [:ip_range, :reverse_dns, :name_dns, :add, :confirm_nodes, :ip_dns, :save!]. Which is every currently implemented enumeration feature

The run method is called which runs each step. If an exception is thrown, a Swing popup message is displayed. The run method in Runner is straightforward and includes the code for executing the Procs created to write to the interface.

```
def run gui_writer
    arg=nil
    @features.each do |feature|
      writer = gui_writer[feature]
      writer[:begin].call unless writer.nil?
      arg=Enumeration_Runner::send(feature,arg)
      writer[:end].call unless writer.nil?
    end
  end
```

Listing 7: Running the Enumeration Scan

Discovery

The implementation of discovery attempts to find out what ports are open and closed, and if open, what service is running on it. Further attempts are made to not only discern what OS is running, but to check each open port to help determine if there are several machines behind a router or switch with the target IP address. This will assist in future network mapping work. The difficulty is to get a fairly reliable OS scan, there needs to be one or more closed ports on the machine. This presents a problem for getting reliable results since any data returned from trying to access a closed port may be for a different machine if port forwarding is being utilized. Reliable results mean not only accurately determining the operating system, but only returning that operating system. Unreliable scans often give a large list of possible operating systems or just a generic name for the OS, such as Windows or Linux. These sorts of results have little use. To be able to successfully attack a system or find a list vulnerabilities these general results are not very helpful.

A future workaround may be to use passive scanning. This implementation uses nmap exclusively and therefore only offers active scanning. Another workaround is to closely look at the results of a service scan. Quite often web servers and SSH implementations are verbose and will tell what OS they are running on.

This initial implementation leaves more work for the plug-in writer because the underlying libraries assume a single port. This puts the responsibility of dealing with lots of results from many ports on a list of nodes on the plug-

in writer. This may allow for more flexibility, at the cost of more work. An alternate way of dealing with this is by passing in an array of Nodes, and also a block or Proc detailing the scan and how to save the data. This is the closer to the "Ruby way" to do it. It is similar to method used in Enumeration, except instead of static methods called dynamically according to the steps in the list, *anonymous functions* are used. Since blocks and Procs are *closures*, this may give even more flexibility in writing plug-ins.

Writing information is handled differently as well, instead of storing the messages in Procs, all output is written in the discovery method in the plug-in. This simplifies the libraries, at the cost of not being able to give messages about any problems or events of interest in the libraries. Information about starting and completing scans on each node are about the most detailed information that can be given in the plug-in itself.

The library that performs the scan has facilities to dynamically run different types of scans. Because the results returned can vary greatly depending on the scan type a system was developed to easily add a custom parser. The plug-in writer does not have to write the parser, but easily could be done using a block. The parsers are part of the external Nmap library. Currently, the parser code is an if-else structure. This is fine since only a few scans are currently supported, but each parser should be separated out in their own classes or modules eventually. The only exception is the service parser which is in its own class. This parser doesn't need to change over the various scans.

Since one of the goals of this project is to lessen the learning curve of learning these tools, and with Nmap learning the command-line arguments is a big part of this curve, the type of scan is called by method name which handles getting the correct arguments to the program that controls the Nmap scans. However, it does not relieve the plug-in writer from knowing what scan to use in each situation, which is valuable knowledge. To implement this, a dynamic facility in Ruby is used: method_missing. What this method does is that it gives a way to properly handle a call to a method that does not exist. This allows methods

to be dynaimcally created to fit a specific need during runtime. To simplify this process a class called Scanner was created that actually contains the command line arguments needed to run the requested scan. Each method assigns the arguments to an instance variable. This is not necessary, and could be easily done in method_missing, but this was a good way to make supported method names and argument lookups easy to find and understand. The method names follow the convention "scan-type"_"options"_scan. For example: syn_scan for a simple scan that uses syn packets, or syn_service_scan which is a syn scan that also tries to figure out what service is running at the open port.

The supported scans also have optional arguments that are not initially included. These options include timing and stealth to help reduce scan time and to try to evade intrusion detection systems. They are not included to make evaulation less complex and error prone. TCP scans are straightforward because of the three-way handshake required to set up a connection. If the port is open, the protocol guarantees that a response will be received. UDP does present a serious problem. The protocol makes no guarantees of a response. As it is a connectionless protocol the three-way handshake does not occur. This not only makes UDP results inconsistant, it makes the scans last much longer. A default UDP scan can take hours and a complete scan may take days. Because of the time required for a UDP scan the prototypeaudit only tests three UDP ports.

To assist users with understanding tool output, feedback via the interface is used. All messaging is done by the audit script. Because of this the size of the discovery script is very large. For brevity messages meant to be written by the GUI are removed from the listing below.

```
1   nodes=@scan.nodes
2
3   scans = Discovery_runner::Scanner.new
4   nodes.each do |node|
5     results = []
6     node=node.node
7     run = Discovery_runner::Runner.new node.ip_addr,scans
8     results << run.syn_version_scan(ports)
9     results << run.udp_version_scan(" -p161,162,2049 ")
10
11    results.each do |res|
```

```
12
13          closed = res.closed
14          open = res.open
15          os=nil
16          unless closed.nil? && open.nil?
17            if open.size > 0 and closed.size > 0
18              os = OS_scan.run_port(node.ip_addr, res.open, closed
                    [0],get_message)
19            end
20          end
21
22          open_ports = res.open
23          closed_ports = res.closed
24          open_filtered = res.respond_to?(:open_filtered) ? res.
                open_filtered : []
25          services = res.service
26
27          open_ports.zip(services) do |port,service|
28            db = Db_port.new node.id,port,'open'
29            db.services service unless service.nil?
30            db.os_types(os[port]) unless os.nil?
31            db.save!
32          end unless open_ports.nil?
33
34          unless open_filtered.nil? or open_filtered.size == 0
35            open_filtered.each do |port|
36              db = Db_port.new node.id,port,'open|filtered'
37              db.save!
38            end
39          end
40
41          unless closed_ports.nil? or closed_ports.size==0
42            closed_ports.each do |port|
43              db = Db_port.new node.id,port,'closed'
44              db.save!
45            end
46          end
47      end
```

Listing 8: Discovery Scan

Line 3 creates the object holding the names of the supported scans. Line 4 iterates over each node. As shown, the current node is scanned and the results saved before moving on to the next node. A popup giving the option of what ports to run is given, if nothing is entered the default 1000 port Nmap scan is run. Line 8 is the first example of using method_missing, syn_version_scan does not exist in the Runner class. The Runner class will be listed in its entirety.

Next the UDP scan is run. As noted, only three ports are tested and is hardcoded. In a robust audit, the same ports, typically as off them, would be run for both TCP and UDP scans. At this point, the results of both scans are saved in a list. This list is now iterated over in line 11. At this point, a list

of open and closed ports are made and this information is used to determine whether to run a scan to check for the operating system running.

The Nmap parser returns the results in a hierarchy of objects which is where methods such as open_filtered() come from. On line 24 of the above code a method call respond_to? is called. This method returns true if the argument passed in exists as a method. If a scan didn't find a particular set up data, the object or methods would not be created. This is a tradeoff when implementing dynamic code. The remaining code sets up the data in the proper node, port and service wrapper classes and then saves the data to the database.

The following code shows the Runner class in the Discovery module.

```
1  class Runner
2    def initialize node,scan
3      @node=node
4      @scan=scan
5    end
6
7    def method_missing name,*args
8      super if !@scan.respond_to? "#{name}"
9
10     arg = @scan.send "#{name}"
11     args.each {|a| arg += "#{ a}"} unless args.nil?
12     scan = Port_scan.new(@node,arg)
13     scan.run
14     scan.parse @scan.type
15   end
16 end
```

Listing 9: Discovery::Runner class

The object initializer has two arguments: the node to be tested and a reference to the Scanner object. The reason it is passed in and not created in Runner is to allow the audit writer to create his own class if it is necessary. The first argument in *method_missing* is the name of the method as a string. The interpreter passes this name from the method call. The second method is an array of arguments passed in as if each element was a method argument. The first line ensures that the method name is supported. If the method does not exist an exception is raised via the overriden *method_missing*. The next line uses the method name to call the Scanner object to get the appropriate Nmap argument. The next line iterates over any passed arguments and appends it to

the Nmap argument. The scan is then run, parsed and the results returned.

This code will support any and all scan types that can be implemented. As noted before the Scanner class really isn't necessary, it could be worked into *method_missing* as a series of if statements, but the way it is implemented is easier to understand.

In future work, saving data should be placed in a library as it is fairly complicated. The reason is that the database only saves certain service states to save space. For basic TCP scans, the three possible states are open, closed, and filtered. Open is interesting and because certain information can be ascertained by the response that signifies a closed port it is also added to the database. Filtered in itself is not interesting, and would do nothing but potentially add tens of thousands of database entries to a single network scan. There are also some special cases that need to be addressed in more details such as the UDP Nmap state open—filtered.

Vulnerability Testing

There are many possible types of vulnerability scans that can be performed. Passive scanning, querying online vulnerability databases, using programs that test for known vulnerabilities such as Nessus, and more active methods that can find both known and unknown vulnerabilities via brute force methods like injecting shellcode or running fuzzers. Each method can be extremely complex.

There are two implemented tests in NetProbe. The first is querying online vulnerability databases, more specifically the Open Source Vulnerability database(osvdb.org), It uses a simple format for programmatically submitting queries. It is in the form osvdb.org/api/¡API KEY¿/args and is run using the HTTP class in the Ruby library as it is much simpler than the Java version. The API key is assigned after registering, and grants 100 requests a day by default. The key currently being used has no limit which was granted by the OSVDB administrators.

Another way to access the data is to download the entire database, which

solves the limited access problem and also simplifies access as requests no longer have to be HTTP requests. The downside is that the database is large and it changes often requiring downloading the entire database several times a week.

The current implementation assumes HTTP access. Since new vulnerabilities can be added every time there are no request savings by caching the results over time, however on any given scan, results can be used to optimize the number of requests if there is the same service running on multiple nodes in the same network.

The second vulnerability test that has been implemented for the prototype makes use of the web vulnerability scanner called Nikto. This program is used to ferret out various levels of information about a web server. It can also be used to different degrees in testing out web applications. It also supports HTTPS. There are various levels of configuration, from the simple such as testing different ports to more complex ones like CGI directives. For the default scan the default scan, ports 80 and 443 are tested on every node in the scan when they are open. Unlike the OSVDB tests, only one test per node is run. The reason for this is that both ports are typically bound to the same web server, this saves a lot of time without loss of information.

Vulnerability Testing Implementation

The implementation of the prototype mimics a *domain specific language* (DSL) approach which results in very little code being need to be written in the plug-in. To mimic a DSL, the methods used to run the scan are dynamically created and placed in the Kernel module. In a real DSL setup, these methods would likely be created in a separate module file which is then read in like a text file and then treated like code. The reason it is not done this way is that there are several complications to dynamically reading the DSL module when the rest of the scan is not done this way.

Both the OSVDB and Nikto scan controllers are placed in the Kernel module. However, The classes that do the actual work are not dynamically created, but

are directly accessed by the new methods in Kernel.

```
lambda {
  messages={}
  scan=vuln_scan
  Kernel.send :define_method, :vuln_scan= do |s|
    scan=s
  end

  Kernel.send :define_method, :vuln_scan do
    scan
  end

  Kernel.send :define_method, :message= do |msg|
    messages[msg.type]=msg
  end

  Kernel.send :define_method, :clear_messages do
    messages.clear
  end

  Kernel.send :define_method, :writer= do |gui_writer|
    writer=gui_writer
  end

  Kernel.send :define_method, :run_generic_osvdb_service_scan do
    scan.nodes.each do |node|
      ports=node.ports
      ports.each do |port|
        if !port.service.nil? and !port.service.product.nil?
          writer.appendText "Running OSVDB search for #{node.
              ip_addr}:#{port.port}"
          GUI_utils.progress_runner writer do
       s=Osvdb_scan.new port.service.product,writer, port.port
       next unless s.scan
       s.parse
       s.save! port.service
          end
        end
      end
    end
  end

  Kernel.send :define_method, :run_nikto_scan do | cgi_dir|
    scan.nodes.each do |node|
     ports=[]
     node.ports.map {|p| ports << p if  is_http_open_port p }
     next if ports.nil? or ports.size == 0
     port_str=''
     ports.each {|port| port_str+=port.port.to_s+' '}
     writer.appendText "Scanning #{node.ip_addr} ports #{port_str
         }"
     GUI_utils.progress_runner writer do
      n=Nikto_scan.new node,ports, cgi_dir
      n.scan
      n.parse
      n.save!
     end
    end
  end

  Kernel.send :define_method, :is_http_open_port do |port|
```

```
59        (port.port==80 or port.port==443) and port.status=='open'
60      end
61
62    }.call
```

Listing 10: Vulnerability Scan DSL

Each method adds either utility functionality or a vulnerability scan. This method is like the flattened scope used to store the reference as a controlled global. The OSVDB and Nikto scans are standard object-oriented scans. The cgi_dir argument in the run_nikto_scan method allows for a variety of configuration options. Each scan type returns an XML file which is parsed as a hierarchy of objects and then stored in the database much like the Nmap results.

While each HTTP(S) port is accounted for seperately in the database, all of the discovered HTTP(S) ports in a single node are run using Nikto at the same time. This approach makes the audit script very concise and greatly increases the speed of the Nikto scan without sacrificing accuracy. Again, all code that produces output to the GUI is initially produced in the script and is omitted from the below listing.

```
1 OSVDB.osvdb_key= Vulnerability.osvdb_key
2 Vulnerability_Scan::setup scan_ref
3 Vulnerability_Scan::writer= get_message
4 run_generic_osvdb_service_scan
5 run_nikto_scan nil
```

Listing 11: Vulnerability Audit Script

The OSVDB key is retrieved from the database and passed to the external OSVDB library. The next line invokes the lambda shown above, and the interface writer is passed into the DSL. The last two lines show all that is needed to run a complete vulnerability database and nikto scan. The downside to this approach is that it may decrease the flexibility of the vulnerability scan. But it is a relatively simple matter to add functionality. A new method is added in the DSL to run it, and outside of the DSL methods to run, parse and save the results are needed. It does turn adding new functionality into a short list of steps.

Reporting

The last step in the default scan is documentation and reporting. This is arguably the most important step as it recreates the steps taken and displays the results. It can also act as a guide for further evaluation and as such the report should be in an easy to edit format. It should also be marked up in a format that makes it easy to read and is clear and concise enough to be included in a final audit report to a client. The code used to create this report should eventually be flexible enough to accept data from anywhere, even a standalone test such as the ones that can run under the Tools menu bar item.

The initial reporting code has just enough features to produce a report using data from the default audit already described. The methods used are standard Ruby techniques: mixins and singleton methods. Mixins are a Ruby compromise between multiple inheritance and Java interfaces. It is a module that is included in a class. When a class includes a module, all of its methods are now available in the classes object as if the methods were implemented in the class. It is possible for the methods that were mixed in to be able to access members of the class or object that included them. It is also possible to define object instance variables in these modules that the object that mixes in the module can use as if they were object members. A simple description of mixins are Java interfaces with implementation already defined. This is a very powerful feature that adds a lot of flexibility and does not require much boilerplate code like Java interfaces do. This functionality explains why enumerable classes(Array, String, even Hash) of any type(s) can sort themselves with the same sort function that is mixed in from the Enumerable module. This sort function optionally takes a block which allows for custom definitions of order without writing a new sort method.

Singleton methods are methods that only exist in a given object. These methods are added or opened after instantiation. This method gives the ability for specific audit techniques to have its own reporting code that can be inserted into the report object at run time. This produces a system that will only have the functionality it needs to complete the current task without a lot of

extraneous code cluttering up the report class file. Here is a simple example of a singleton method:

```
1 a=Object.new
2 b=Object.new
3
4 def a.my_method
5    puts "singleton method"
6 end
7
8 a.my_method #prints singleton method
9 b.my_method #raises undefined method error
```

Listing 12: Singleton Method Example

In addition to using Singleton methods, which are built in to the language, the Report class has two methods of generating the functionality needed to prepare and save the report. The first method is using a built in template system. But informing the Report object what file type it is preparing, it can automatically mixin the modules it needs. The second is to manually mixin the appropriate modules. The former method is less flexible, but easy for the audit writer to implement since the templates are already provided. The latter method is very flexible, at the cost of use of use.

The current implementation creates Model objects to directly read from the database and then formats the data into the appropriate file type. Currently, there are two file types supported by the reporting module: text and PDF. This means there is a template for each type. This template handles the actual formatting of the data. For the initial evaluation a PDF is generated, but this ignores many possible uses for the reporting module which will be discussed in detail in future work.

The report class is called Report and is very simple. The report specific parts contains code to setup and save the report. Save is actually done in the specific module that writes to disk. The reason for this is because different file formats might use a third party library whose save function may have a different name then other libraries. This makes functionality that all file types need "cross-format". The rest of the code in Report exists to simplify the process of mixing in modules.

```
1  class Reports
2   attr_accessor :file
3   def initialize file_path, type=nil
4     @file = get_file_type type
5     @path=file_path
6   end
7
8   def save_file
9     save!
10  end
11
12  def self.mixins *modules
13    modules.each do |mod|
14      self.send :include, mod
15    end
16  end
17
18  def add_module *modules
19    modules.each {|mod| self.extend mod}
20  end
21
22  def custom_add
23    if block_given?
24      yield @file
25    end
26  end
27
28  def self.custom_pdf file_path, arr_blocks
29    r=Reports.new file_path, :pdf
30    arr_blocks.each do |block|
31      block.call r
32    end
33    r.save!
34  end
35
36  private
37  def get_file_type type
38    if type==:text
39      Reports.mixins Text
40      Text::setup
41    elsif type==:pdf
42      Reports.mixins PDF
43      PDF::setup
44    end
45  end
46 end
```

Listing 13: Report class

In line 2, an accessor and mutator is created in case a template mixin is not automatically used. The initialization starts at line 3, the path to where the file will be saved is passed in and optionally the file type. The file type is a symbol. Currently, the only two valid symbols ate :text and :pdf. If anything else is passed in, it has the same effect as if nothing were and the file mutator will need to be called manually. Note that references to file and path are created

in ***initialize***. For mixins to be able to use them they must follow the naming conventions for the mixed in methods to be able to use the object references.

Lines 12 and 18 define the two mixin helper methods. Normally, a module is mixed in statically. The problem with this method is that all objects will have the same functionality, which is something our report class does not want. The goal here is to define a common interface without having to repeat a lot of code, or have many different types of reporting classes that all have different method names and arguments.

The first method, ***mixins*** is a class method that takes a variable number of arguments. Each argument must be a module. The method iterates and includes each module. This is similar to the normal way of including modules. Once the module is included the methods are available across all instances. If only one report type is going to be generated this is an ideal method, and is the assumption of the initialize method when a type is given. It is possible to remove the mixed in methods via a library called ***mixico*** or by removing them manually, neither is currently implemented.

The second method, ***add_module*** also mixes in modules, but does it in a way that makes the mixed in methods Singleton methods. This is the preferred method when multiple types of files need to be generated and will be more useful in the cases where the audit writer prefers more control over ease of use.

The next two methods are convenience methods. The method ***custom_add*** allows for passing in a block that can use the reference to file to write or format it in some way. The other method***custom_pdf*** is simply another way to create the PDF, this time just using an array of ***Proc***. A use for this is if methods of each audit step were created to handle its own reporting, this functionality could return Proc's to be handled by Report later on.

Since the prototype audit generates a PDF, the PDF module will be described in detail. This module is mixed into Report.This module uses the Prawn library which is the defacto standard Ruby PDF library. It is very complex, but this module simplifies it to meet the fairly simple needs of this project by pro-

viding default layout information.

```
module PDF
 def self.setup
  Prawn::Document.new
 end

 def header text
  @file.text text, :size=>20, :align=>:center, :style=>:bold
 end

    def paragraph text, indent_level=0
      @file.indent(20*indent_level) do
        @file.text text , :size=>10
      end
    end


    def table data
      @file.table data,
        :position => :left,
        :column_widths => { 0 => 100, 1 => 400},
        :border_style => :grid,
        :vertical_padding => 10
    end

    def custom text, size, align, style
      @file.text text, :size=>size, :align=>align, :style=>style
    end


    def newline lines=1
      @file.move_down(lines*@file.font_size)
    end

    def save!
      @file.stroke
      @file.render_file @path
    end

    def horizontal_rule width=0.5
      @file.line_width=width
      @file.horizontal_rule
    end

    def horizontal_line x1,x2,width=0.5
      x1=0 if x1.nil?
      x2=@file.bounds.width if x2.nil?
      file.horizontal_line x1, x2, width
    end
end
```

Listing 14: PDF Module

The method ***setup*** simply instantiates Prawn and returns the reference. Other than the save! method, the rest of the methods provide a wrapper around a common PDF element, such as header, paragraph, table and horizontal lines. For example, to create a table, call the method ***table*** and pass in an array that

describes the table layout and data. To draw a horizontal line, call ***horizontal_rule*** with an option width value. There is also a method to set custom text attributes. If more advanced control over the file is needed during runtime, the PDF module can be opened or other modules can be mixed in.

This module contains no file data details, only attributes. The actual data can be generated by the audit writer or through the use of predefined templates. Using predefined templates is easiest, but it is difficult to cover all possible database models and plugin functionality. Using the dynamic functionality described in the **Report** class a mix of both approaches could be used. The current implementation uses a predefined template. This class creates a **Report** instance, fetches the data from the database and controls the writing procedure. Listing 15 shows the code that does this.

```
1  module Reporting
2
3    class Default_PDF_Report
4      def initialize path, db
5        @report = Reports.new path, :pdf
6        @db=db
7      end
8
9      def runner
10       heading
11       basic_info
12       nodes=@db.nodes
13       nodes.each {|node| node_data node }
14       @report.save_file
15     end
16
17   private
18     def heading
19       @report.header "Network Topology and Security Evaluation"
20       @report.custom "Generated by NetProbe", 16, :center, :bold
21       @report.custom Time.now.strftime('%b %d, %Y'), 14, :center, :
             bold
22       @report.newline 3
23     end
24
25     def basic_info
26       @report.custom "This report details the results of a #{@db.
             auditprocess.name} scan against the network
27 #{@db.target}.
28       This network has #{@db.nodes.size} node#{'s' if @db.nodes.
             size >1} that #{@db.nodes.size==1 ? 'was' : 'were'}
29 scanned.", 10, :center, :italic
30       @report.newline 3
31       @report.paragraph 'The following tables break down the
             results of the scan.'
32       @report.newline 3
```

```
33      end
34
35      def node_data node
36        data = []
37        data << [:text=>"Node Address: #{node.ip_addr}",:colspan=>2,
              :align=>:center]
38        node.ports.each do |port|
39          data << ['Port',"#{port.port} "]
40          data << ["Status", port.status]
41          service data, port.service, port.port unless port.service.
                nil?
42          #os_type data, node.os_type unless port.ostype.nil?
43        end
44        nikto data, node.nikto unless node.nikto.nil?
45        @report.table data
46        @report.newline 5
47      end
48
49      def nikto data, nikto
50        data << [:text=>"Nikto Results", :colspan=>2, :align=>:center
              ]
51        data << ["Server Data",nikto.server_info]
52        data << ["SSL Ciphers",format( nikto.ssl_ciphers)]
53        data << ["SSL Issuers",format( nikto.ssl_issuers)]
54        data << ["SSL Data",format( nikto.ssl_info)]
55        data << [:text=>"#{nikto.niktoitems.size} potential issues
              found.", :colspan=>2, :align=>:center] if
56  nikto.niktoitems.size > 0
57        nikto.niktoitems.each_with_index do |ni,index|
58          data << [:text=>"Issue #{index+1}", :colspan=>2, :align=>:
                center]
59          data << ["OSVDB ID","#{format( ni.osvdbid)} "]
60          data << ["OSVDB Link",format( ni.osvdb_link)]
61          data << ["Description",format( ni.description)]
62          data << ["Issue Link",format( ni.name_link)]
63        end
64      end
65
66      def service data, service, port
67        data << [:text=>"Service Information for Port #{port}", :
              colspan=>2, :align=>:center]
68        data << ["Name",service.name]
69        data << ["Version",service.version]
70        data << ["Product",service.product]
71        data << ["Operating System",service.os]
72        data << ["Owner",service.owner]
73        data << ["Other",service.other]
74        return if service.vulnerabilities.nil? or service.
                vulnerabilities.empty?
75        vulnerability data, service.vulnerabilities
76      end
77
78      def vulnerability data, vuln
79
80      end
81
82      def format data
83        (data.nil? or data.size==0) ? "None" : data
84      end
85
86    end
87  end
```

Listing 15: PDF Reporting class

The methods ***initialize*** and ***runner*** setup and control the operation. All of the rest of the methods fetch the data and call the appropriate methods from the **PDF** class. As long as this class is kept up to date with the database models, creating PDF reports for any audit plugin is simple.

```
1 def report
2     path="reports/#{scan_ref.target}.pdf"
3     report=Reporting::Default_PDF_Report.new path,scan_ref
4     report.runner
5     Apprunner::runner_close_all('acroread',path)
6   end
```

Listing 16: Report class

There are several ways to get the path, in this case it is simply hardcoded and named after the target that was entered at the start of the scan. The appropriate reporting class is instantiated and then run. The last line is optional. ***Apprunner*** is an external library that provides basic external program execution capabilities. The method ***runner_close_all*** starts the given program with the path to the PDF file and then automatically closes the program's output, input and error pipes. In this case the method call automatically opens the file using Acrobat Reader.

Examples of generated reports are contained in the appendix.

Targets

The evaluation will consist of running the automated and manual scans against two targets. The first is maplewoodsoftware.com with kind permission from Maplewood Software President John Janzen and the other is mccombsonline.net. The former target consists of multiple IP addresses over three non-continuous IP ranges and on two physically separate networks. The latter target is a single IP address pointing to a virtual machine.

Manual Audit

The manual audit will mirror the steps and functions of the automated audit using the same programs. Each step will be documented and the results saved in PDF format when possible, but otherwise plain text(e.g. .txt or .xml). No additional scripting will be used. That is, each program run will be run with input manually typed in. The input settings of each tool will mirror that of the automated audit. If a default in the audit script is lacking or running manual scans is easier to alter default, it will be noted in the results section.

Automated Audit

The automated audit will cover all the functionality previously detailed as implemented and results saved as a PDF file. The default audit script has several defaults, some of which can be altered during runtime. When default exist to save time in scans it is only for the purpose of simplifying the report and making demonstrations feasible. Production quality scripts will not use this rationale for configuring default settings. The default behavior for each step is as follows:

- Enumeration
  - If a *CIDR* is given, the addresses are calculated based on the given IP address or resolved address of the URL.
  - Each address is run through a reverse DNS scan.
  - A popup displaying the URL's are shown with the option of viewing their whois record and removing them from the scan
  - There is no user defined settings. The steps taken are entirely dependent on the type of data that is discovered.
- Network Discovery
  - Nmap scans are run against the defined TCP default ports for Nmap or the user defined settings.

  - A option to manually set ports during runtime is included.
  - Only 3 UDP ports are scanned. This is because UDP scans can take hours or days. No user intervention exists yet.
  - Service scans for all open ports are run.
  - If a node has at least 1 open and 1 closed port an operating system detection scan is run.

- Vulnerability Scan
  - An OSVDB scan is performed on all open ports where the service has been found.
  - The scan settings are currently hard coded and can not be altered during runtime.
  - A nikto scan against nodes that have ports 80 or 443 open is run with no other options enabled.
  - While the Nikto scan library developed does have the ability to accept other arguments to alter the Nikto scan, there is currently not a runtime option to do so. This is simply because these options can increase testing time to several hours or days.

- Reporting
  - A PDF of all information generated in the scan is produced and saved with the format ¡target¿.pdf and saved to the folder reports/. An option to change this has not been included.
  - There is a runtime option to view the document using the system's default PDF viewer.

Evaluation Criteria

The main criteria used to compare both types of audits are as follows: The *accuracy* of each audit. The IP ranges, services and operating systems running

on both networks are known beforehand. Given the the same input each tool used will provide the same output. Because of this, the ease of managing all the information found in previous steps is the most important part of this criteria. *Ease of Use*, this is a subjective criteria but will be based on how long it takes to get setup and manage the audit details. The *quality and usefulness of the generated reports*. A comparison of the detail and quality of the reports generated. *Extensibility*, an evaluation on the difficulty of extending the audits to include more details and coverage. All tests will be run with the default settings.

Conclusion

This section details the results of the manual and automated audit.

Results

The reports for both scans done with NetProbe are included in the appendix. The reports for the manual tests are included separately in a folder. The reason for this is that each step produces a separate document, and may be larger than the generated reports from NetProbe.

The first target for Netprobe is mccombsonline.net As previously mentioned this target has one physical node to test. As such it is very straight forward. The enumeration step found 4 URL's, with only one valid URL (*Figure 6*). The other three belong to nameservers and the gateway from the host provider RailsPlayground. These three URL's were discarded. Discovery was run with the default port, version scans, and OS detection. The vulnerability scan was performed on all open ports and Nikto tested ports 80 and 443. There are two more HTTP ports, 7777 and 7778, but as mentioned this is not accounted for in the initial implementation. In this case, doing so is redundant as these two extra ports point to the same webserver that is used for 80 and 443.

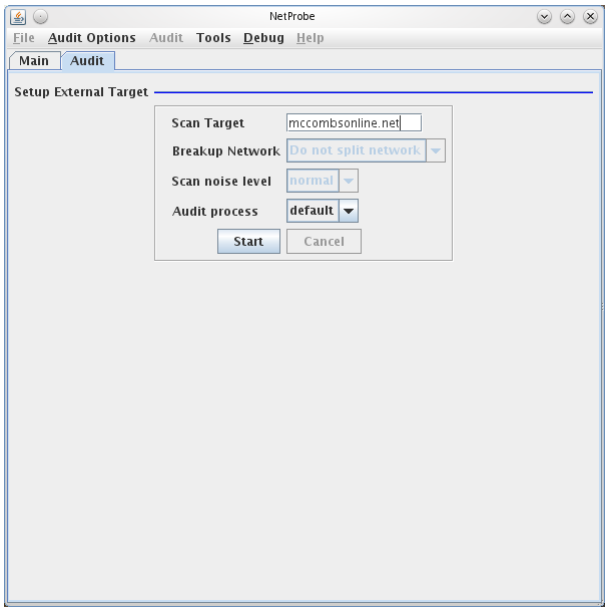

Figure 5: Initializing Audit

The manual scans against mccombsonline.net were identical except for one area: vulnerability detection. Since the vulnerability scans for OSVBD in Net-

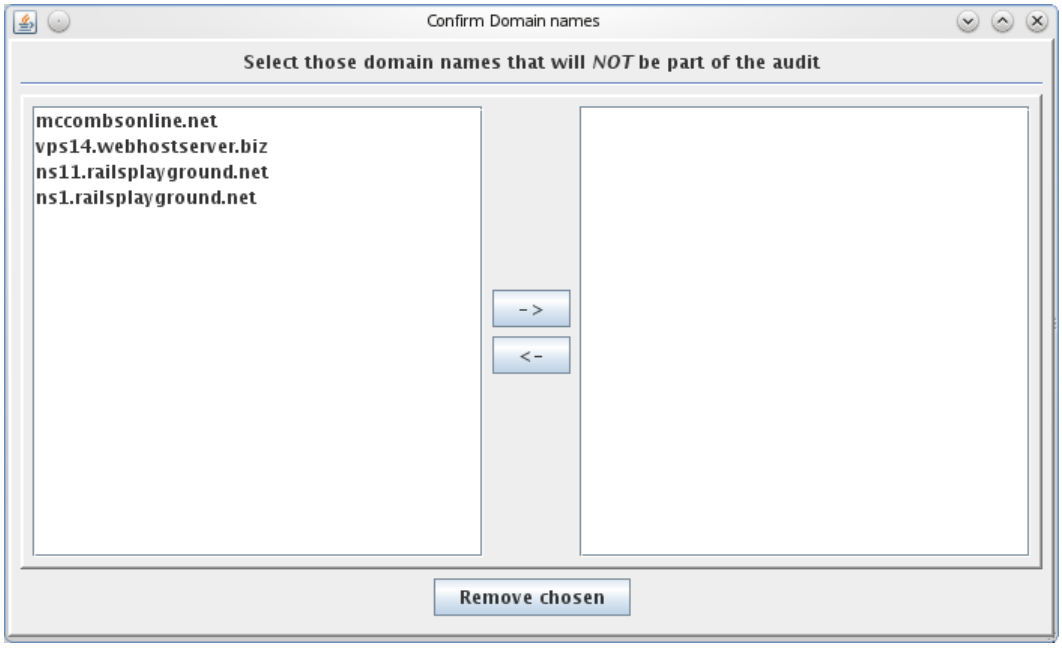

Figure 6: Removing Invalid URL's

probe are static given a vendor and product, it will always produce the same results. However, when querying OSVDB manually via a web browser, it is trivial to expand and narrow searches. This discrepancy will be discussed in further detail in the future work section.

Against a single target there is little advantage to Netprobe other then not having to manually run programs at the command line. This is of little concern since most security audits are run against networks with multiple targets and sub-networks. The main disadvantage of less granular control with regard to vulnerability scans is critical. Since this step does not confirm the existence of a security problem it is less critical than other steps that have more granular control built in, such as port scanning.

The last test is against the networks of Maplewood Software. The network IP ranges are known. The structure of the networks have changed since working there. This makes this test closer to a real world audit, since not all variables are known before hand. The target that the audit is given is simply maplewood-

software.com. The IP ranges are 206.63.184.10/29 (206.63.184.8-206.63.184.15), 206.63.184.67/28 (206.63.184.64 -206.63.184.79, and 71.39.195.145/29 (71.39.195.144-71.39.195.151). The first address in each range is the gateway address that belongs to the ISP, and is not scanned. The last IP is the broadcast address so it not scanned. The existence of an IP address in the network does not imply that there is a machine connected to it. In this case most of the addresses are not.

The automated test found 4 IP addresses with a machine connected to it. The scan did miss one: 206.63.184.11. In this case, coverage remained at 100% because this IP address is bound to the same NIC as 206.63.184.10 and each reports the same services. Several invalid URL's were removed from the test that were the gateway addresses to the network. The scan was run with the default settings. It took about 15 minutes to complete.

The manual test against Maplewood Software was much more difficult than the first manual test. An Nmap list scan was performed to resolve maplewood-software.com. This gave the address 206.63.184.67. The low IP address of this address range was determined by decrementing the IP address by one and running a reverse DNS scan until an invalid address was found. The process was repeated for finding the upper bound. This process is time-consuming and not necessarily accurate. The other ranges were found by performing a DNS look-up for mail servers which were reported at 206.63.184.10 and 71.39.195.148. The latter address no longer has a mail server attached, but the DNS records have not been updated to reflect this. After the ranges for each network was set, an Nmap scan for each node following the defaults for NetProbe were followed and saved to a XML file if any open or closed ports were discovered. A service scan was performed at the same time. Nodes that have at least one open and one closed port had a OS scan performed for each open node. Again results were saved for all nodes reporting services and operating systems found.

A manual search of osvdb.org was performed for each known service running. This is more accurate when done manually as previously mentioned, however results either have do be written down manually or the HTML pages saved. A

Nikto scan was performed with default settings. As expected the results were identical, and the only difference is that automated scan was able to process the results faster. A report was not written, but the report would either be written up as a summary or all the generated files would serve as the report. This is time-consuming and can be inaccurate because of the need to manually parse through potentially hundreds or more files that may not be in easy to read format (XML, CSV).

A critical problem with the automated scan is that given the target, maplewoodsoftware.com, several potential IP addresses were missed. In this case, it is not an issue, because those IP addresses have no machine associated with it. However, this is a problem because that will not be the case everywhere. The problem is that the default address doesn't handle finding *CIDR's* very well as well as nodes that do not have a URL associated with it. This will be addressed in future work.

The major downside of the manual scan is all the data that needs to be manually processed. It is easy to omit nodes or open ports to scans and reports. It is also much more time consuming as multiple scans for each target need to be manually configured and run.

While the automated audit is currently flawed, it has promise. Issues such as adding more user control, handling network enumeration and vulnerability scans more efficiently and accurately needs to be addressed as high priority issues before a simple default audit can be considered useful in production.

Software Design Results

The software design techniques all have value. However, the ones that cause the actual audit plugin to require little coding are the "winners", provided they can be adapted to any situation and remain flexible. This will be discussed in further detail in future work. These techniques were shown in the vulnerability and reporting sections of the scans. When development continues, a mix of the two will be used for the entire project, other than the user interface. Each

large area of functionality, such as enumeration, will be broken down into small modules that perform a single task. Each module will have certain prerequisites that need to be complete before running the module. For example, before a nmap service scan can be run, the module that determines if a port is open needs to be run, and that module needs to know what IP addresses to scan, which requires other modules.

What this will lead to is a system that is very similar to off the shelf component development, except that the components do not have to be so generic. They can be written for this system, and thus have knowledge of the models and views. It will also make it possible to have an audit generator that can write the audit script after the user selects the audit functionality he wants to use. This of course, leads to not needing to know any programming to construct custom audit scripts.

The use of ActiveRecord for handling the models will stay intact. What needs to change is that each module should know about only its relevant models and be able to produce output for the reporting module to execute. An example is the OSVDB vulnerability module(s) will know how to create its own data stream, but it must also be able to tell the reporting module that each stream belongs to a certain node. The responsibility of the actual formatting of the document still belongs to the reporting module(s). How this will be accomplished is still unclear and will need much more up-front design.

The interface will remain a graphical user interface, but other GUI libraries should be considered. Swing is very verbose and fairly buggy. The MVC architecture will remain. However, the bindings between them should be generalized more to keep each concern separated. This can be done by creating a DSL that mimics Ruby on Rails. That is, common functionality, such as connecting the controller with the view, should be distilled into libraries that handle a lot of the work. With this method, there will not be strings created that get written to the user interface in the controller sections of the program. The controller section is considered to be the audit script itself and user interface listeners.

Rewriting the user interface to behave more like a web browser, in terms of displaying interface elements will be needed to be done to simplify the process of creating the DSL.

Future Work

There is nearly an infinite number of features that could be added to the project. The first thing to be done before the project is ready for expansion and as a possible project for students to work on is refactoring the architecture as discussed in the software design results section.

Once the refactoring is complete, the project can be extended in multiple ways by concurrent development projects. The remaining part of this section will briefly discuss possible additions and bug fixes for current issues. Each extension should be developed with the project goal in mind: to produce a program that leverages existing tools and libraries that automates and simplifies the security audit process. Given that, there are several reasons to create new tools and libraries or extend them before bringing them into the project. The first is that a tool may be lacking functionality. An example is a packet sniffer library that does not directly support an application or network layer protocol. Existing tools may be flawed, and in these cases it may make sense to create a new tool or fix the existing one. There may be libraries or programs that could work together so creating middleware that links these programs or libraries can be useful. An example of this is combining an existing client HTTP library to a fuzzing library and an ARP poisoning program to create a customized web proxy to not only conduct fuzz testing but to create actual exploits such as *man in the middle*.

The initial implementation of NetProbe did not leverage one of the advantages of using the Java Virtual Machine. This advantage is the massive networking libraries that exist for Java. The official API only has network libraries for common tasks such as basic client/server sockets, HTTP client socket and so forth. There is not support for specific low level protocols and application

protocols such as FTP or mail clients. Even the HTTP client libraries are fairly limited. Third party libraries are another matter. These often extend the basic services that the official API provides and these should be leveraged. An example are the Apache client and server libraries. They have support for full HTTP support including managing cookies and secure connections. The Java API has these, but are not integrated like the Apache modules. They also implement other application layer protocols for client and server such as FTP, POP, and even complex protocols such as ActiveDirectory. These libraries were not written for security usage but can be leveraged to assist in testing.

The current implementation needs work. An issue with the vulnerability testing is that all queries are hard coded and can be brittle. During testing the hard coded query for the Apache web server started to produce HTTP 500 errors and needed to be changed in code. These sorts of problems can be solved in 2 ways. The first is dynamic and user defined query generation when an error is returned from the server or no results found. The second is to allow the option of downloading the entire OSVDB database. The latter solution does have the advantage of more powerful search options without running the risk of surpassing the set daily query limit. The trade off is risk of not being up to date and requiring large downloads one or more times a week. Both options should be made available.

Enumeration needs to be reworked as discussed. This is a high priority issue and should be completed before the project is considered robust enough for extending it and for production. The enumeration modules need be able to accurately ascertain all of the IP address ranges, especially in regards to nodes that do not have a URL assigned to its IP address. This is critical, not only for the sake of complete test coverage, but for legal reasons. Auditors can only legally test addresses belonging to the entities they contract with. Testing nodes not belonging to the company or organization they contract with can bring civil or criminal penalties.

Recently, the GUI library QT had its licensing expanded. It now offers li-

censing under the lesser general public license (LGPL). What this means is that the choice between releasing under GPL or having to pay a significant amount to get licensing freedom no longer exists. Under the LGPL, NetProbe can be released under just about any license including proprietary and academic licenses. The library is also much more powerful and streamlined when compared to Swing. There are both Ruby and Java bindings to this library, although with JRuby only the Java bindings can currently be used. Many advances in this area have been implemented recently, but using C code via Ruby libraries is still not reliable enough.

Licensing issues needs to be addressed. Many of the programs that NetProbe utilizes are licensed under the General Public License(GPL). This is a legally binding document, and the precise legal implications are beyond the scope of this paper. However, the examples laid out in the GPL FAQ show that if you make GPL licensed software part of your program, the entire program must be licensed as GPL. This may be problematic if proprietary programs or libraries are also used. However, NetProbe doesn't depend on these programs, they merely run them and read the output, much like an operating system. The author of the GPL, Richard Stallman has clarified that programs that stay "arms length" from GPL software do not need to be GPL'ed.[4] To date, the libraries and programs that were used that are not GPL, but do use licenses that are compatible, such as the Apache license and the LGPL. When the project nears the point where it is ready to be released these issues need to be revisited with legal counsel.

Acknowledgments

I would like to thank Carol Taylor and Steve Simmons for their invaluable advice and support. I would also like to thank John Janzen of Maplewood Software for allowing me to use and abuse his network in support of testing my project.

Appendix

Installation

To install and use NetProbe, extract netprobe.zip into any folder. Before the program can be run successfully

JRuby must be installed and the bin/ folder placed in the system path. Versions 1.4.x, 1.5.0 through 1.5.2 have been successfully tested. The download and installation instructions can be found here: http://jruby.org/getting-started

The following dependencies need to be installed and added to the system path so NetProbe can run them as shell applications. Many of the network system programs are available in Windows via Cygwin. Linux is the recommended OS.

- Host
- Dig
- Whois
- Nmap
- Nikto

Next, Ruby libraries that are used need to be installed via Ruby Gems. These are installed by: jruby -S gem install ¡gem name¿ -v¡version¿. The version is optional in most cases, but not in the cases of the Rails libraries. The gems that do not have an optional version are denoted by a specific version number.

- activerecord 2.3.5
- nmap-parser
- jruby-openssl
- activerecord-jdbcmysql-adapter
- jdbc-mysql

- rake 0.8.7
- railroad (optional-used to create diagrams)
- rails 2.3.5 (only if railroad is used-includes activerecord and rake)
- whois
- net-dns
- xml-simple
- prawn

MySQL needs to be installed next. Version 5.x is sufficient. Installation can be accomplished via the Linux distribution package manager or through: http://www.mysql.com/. Create a schema, currently called "thesis". You can optionally create a specific user account and password for the scheme or use the default root user with no password. If a different database or non-default user is used the file config/database.yml must be edited.

```
1 development:
2  adapter: jdbcmysql
3  database: thesis
4  username: root
5  password:
6 # socket: /var/lib/mysql/mysqld.sock
```

Listing 17: Vulnerability Scan DSL

The socket may need to be uncommented. Another database file config/-database_1, needs to have the same information except the first line is commented out. This is to get around a bug in JRuby. The second file is used to run rake tasks.

Next the database needs to be populated with the required tables and defaults loaded. To do this navigate to the projects root directory on the command line and type: jruby -S rake. This calls the Ruby on Rails migration tool which automatically sets up the database according the ruby files defined in /db/migrate

The program is started on the command line via jruby start.rb. The application needs root access.

Note: There is another way to run this program. All ruby files can be compiled into Java class files and run via a Java Virtual Machine or JRuby. To do this with the JVM compile the ruby files via the jrubyc command. Then run the program as java -classpath "path to jruby.jar" start. This method has not been thoroughly tested with NetProbe as of this writing. The class files can also be packaged as JAR files.

Network Topology and Security Evaluation

Generated by NetProbe

Sep 28, 2010

This report details the results of a default scan against the network mccombsonline.net.
This network has 1 node that was scanned.

A summary of the network scanned:

- 74.63.10.95 Open Ports: 13

The following tables break down the results of the scan.

Node Address: 74.63.10.95	
Port	21
Status	open
Service Information for Port 21	
Name	ftp
Version	
Product	PureFTPd
Operating System	
Owner	
Other	
Port	22
Status	open
Service Information for Port 22	

Name	ssh
Version	3.9p1
Product	OpenSSH
Operating System	
Owner	
Other	protocol 1.99
Port	25
Status	open
Service Information for Port 25	
Name	smtp
Version	1.04
Product	netqmail smtpd
Operating System	Unix
Owner	
Other	
Port	53
Status	open
Service Information for Port 53	
Name	domain
Version	

Product	
Operating System	
Owner	
Other	
Port	80
Status	open
Service Information for Port 80	
Name	http
Version	1.4.13
Product	lighttpd
Operating System	
Owner	
Other	
Port	110
Status	open
Service Information for Port 110	
Name	pop3
Version	
Product	Courier pop3d
Operating System	

Owner	
Other	
Port	143
Status	open
Service Information for Port 143	
Name	imap
Version	
Product	Courier Imapd
Operating System	
Owner	
Other	released 2004
Port	443
Status	open
Service Information for Port 443	
Name	http
Version	1.4.13
Product	lighttpd
Operating System	
Owner	
Other	

Port	993
Status	open
Service Information for Port 993	
Name	imap
Version	
Product	Courier Imapd
Operating System	
Owner	
Other	released 2004
Port	995
Status	open
Service Information for Port 995	
Name	pop3
Version	
Product	Courier pop3d
Operating System	
Owner	
Other	
Port	3306
Status	open

Service Information for Port 3306	
Name	mysql
Version	
Product	MySQL
Operating System	
Owner	
Other	Host blocked because of too many connections
Port	7777
Status	open
Service Information for Port 7777	
Name	http
Version	1.4.13
Product	lighttpd
Operating System	
Owner	
Other	
Port	7778
Status	open
Service Information for Port 7778	
Name	http

Version	1.4.13
Product	lighttpd
Operating System	
Owner	
Other	
Port	161
Status	closed
Port	162
Status	closed
Port	2049
Status	closed
Nikto Results	
Server Data	lighttpd/1.4.13
SSL Ciphers	None
SSL Issuers	None
SSL Data	None
3 potential issues found.	
Issue 1	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0

Description	--- - lighttpd/1.4.13 appears to be outdated (current is at least 1.4.23)
Issue Link	--- - http://74.63.10.95:80/
Issue 2	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0
Description	--- - "ETag header found on server, fields: 0x 0x840009017 "
Issue Link	--- - http://74.63.10.95:80/
Issue 3	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0
Description	--- - "Allowed HTTP Methods: OPTIONS, GET, HEAD, POST "
Issue Link	--- - http://74.63.10.95:80/

Network Topology and Security Evaluation

Generated by NetProbe

Sep 28, 2010

This report details the results of a default scan against the network maplewoodsoftware.com.
This network has 4 nodes that were scanned.

A summary of the network scanned:

- 206.63.184.10 Open Ports: 6
- 71.39.195.148 Open Ports: 0
- 71.39.195.145 Open Ports: 0
- 206.63.184.67 Open Ports: 3

The following tables break down the results of the scan.

Node Address: 206.63.184.10	
Port	22
Status	open
Service Information for Port 22	
Name	tcpwrapped
Version	
Product	
Operating System	
Owner	
Other	
Port	25
Status	open

Service Information for Port 25	
Name	smtp
Version	
Product	Postfix smtpd
Operating System	
Owner	
Other	
Port	53
Status	open
Service Information for Port 53	
Name	domain
Version	9.4.2-P2
Product	ISC BIND
Operating System	
Owner	
Other	
Port	80
Status	open
Service Information for Port 80	
Name	http

Version	2.2.8
Product	Apache httpd
Operating System	
Owner	
Other	(Ubuntu)
Port	110
Status	open
Service Information for Port 110	
Name	pop3
Version	
Product	Courier pop3d
Operating System	
Owner	
Other	
Port	143
Status	open
Service Information for Port 143	
Name	imap
Version	
Product	Courier Imapd

Operating System	
Owner	
Other	released 2005
Port	2049
Status	closed
Nikto Results	
Server Data	Apache/2.2.8 (Ubuntu)
SSL Ciphers	None
SSL Issuers	None
SSL Data	None
6 potential issues found.	
Issue 1	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0
Description	--- - "Number of sections in the version string differ from those in the database, the server reports: apache/2.2.8 while the database has: 2.2.14. This may cause false positives."
Issue Link	--- - http://bones.maplewoodsoftware.com:80/
Issue 2	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0

Description	--- - "ETag header found on server, inode: 466007, size: 91, mtime: 0x4616b2439b800"
Issue Link	--- - http://bones.maplewoodsoftware.com:80/
Issue 3	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0
Description	--- - "Allowed HTTP Methods: GET, HEAD, POST, OPTIONS, TRACE "
Issue Link	--- - http://bones.maplewoodsoftware.com:80/
Issue 4	
OSVDB ID	877
OSVDB Link	http://osvdb.org/877
Description	--- - HTTP TRACE method is active, suggesting the host is vulnerable to XST
Issue Link	--- - http://bones.maplewoodsoftware.com:80/
Issue 5	
OSVDB ID	3268
OSVDB Link	http://osvdb.org/3268
Description	--- - "/icons/: Directory indexing is enabled: /icons"
Issue Link	--- - http://bones.maplewoodsoftware.com:80/icons/

Issue 6	
OSVDB ID	3233
OSVDB Link	http://osvdb.org/3233
Description	--- - "/icons/README: Apache default file found."
Issue Link	--- - http://bones.maplewoodsoftware.com:80/icons/README

The following tables break down the results of the scan.

Node Address: 71.39.195.148

The following tables break down the results of the scan.

Node Address: 71.39.195.145

The following tables break down the results of the scan.

Node Address: 206.63.184.67	
Port	22
Status	open

Service Information for Port 22	
Name	tcpwrapped
Version	
Product	
Operating System	
Owner	
Other	
Port	80
Status	open
Service Information for Port 80	
Name	http
Version	2.2.12
Product	Apache httpd
Operating System	
Owner	
Other	(Ubuntu)
Port	443
Status	open
Service Information for Port 443	
Name	http

Version	2.2.12
Product	Apache httpd
Operating System	
Owner	
Other	(Ubuntu)
Port	1723
Status	closed
Port	5060
Status	closed
Port	2049
Status	open|filtered
Nikto Results	
Server Data	Apache/2.2.12 (Ubuntu)
SSL Ciphers	None
SSL Issuers	None
SSL Data	None
8 potential issues found.	
Issue 1	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0

Description	--- - Apache/2.2.12 appears to be outdated (current is at least Apache/2.2.14). Apache 1.3.41 and 2.0.63 are also current.
Issue Link	--- - http://206.63.184.67:80/
Issue 2	
OSVDB ID	0
OSVDB Link	http://osvdb.org/0
Description	--- - "Retrieved X-Powered-By header: PHP/5.2.10-2ubuntu6.4"
Issue Link	--- - http://206.63.184.67:80/
Issue 3	
OSVDB ID	12184
OSVDB Link	http://osvdb.org/12184
Description	--- - "/index.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000: PHP reveals potentially sensitive information via certain HTTP requests which contain specific QUERY strings."
Issue Link	--- - http://206.63.184.67:80/index.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000
Issue 4	
OSVDB ID	3092
OSVDB Link	http://osvdb.org/3092

Description	--- - "/phpmyadmin/: phpMyAdmin is for managing MySQL databases, and should be protected or limited to authorized hosts."
Issue Link	--- - http://206.63.184.67:80/phpmyadmin/
Issue 5	
OSVDB ID	3268
OSVDB Link	http://osvdb.org/3268
Description	--- - "/icons/: Directory indexing is enabled: /icons"
Issue Link	--- - http://206.63.184.67:80/icons/
Issue 6	
OSVDB ID	3233
OSVDB Link	http://osvdb.org/3233
Description	--- - "/icons/README: Apache default file found."
Issue Link	--- - http://206.63.184.67:80/icons/README
Issue 7	
OSVDB ID	3092
OSVDB Link	http://osvdb.org/3092
Description	--- - "/mw/: This might be interesting... potential country code (Malawi)"
Issue Link	--- - http://206.63.184.67:80/mw/

Issue 8	
OSVDB ID	3092
OSVDB Link	http://osvdb.org/3092
Description	--- - "/sc/: This might be interesting... potential country code (Seychelles)"
Issue Link	--- - http://206.63.184.67:80/sc/

Glossary

Anonymous Functions A function that is defined without being bound to an identifier. It has no name, and may or may not have a reference or pointer.

Backdoor A malicious program that provides the attacker remote access. The program can be any kind of malware.

Buffer Overflow An attack technique that attempts to overrun the allocated memory of a buffer on the stack or queue

Chained Exploit An exploit that requires other programs or nodes to be first exploited in order to succeed.

Classless Inter-Domain Routing (CIDR) - A method of allocating and routing IP addresses that uses y1.y2.y3.y4/x notation, where x is an integer between 0 and 32 to denote the number of leading non-zero bits.

Closure A first class function that has free variables that are declared outside the function but bound to it, as well as the lexical scope of the declaration.

Domain Specific Language A language created to solve a specific problem or set of problems.

Dynamic Typing A type system that enforces type only at runtime. Also called Duck Typing.

Firewall A program that filters packets based on predefined rules that determine whether or not to allow or drop a packet.

Fuzzing A testing technique that uses a mix of crafted and random data to see how the target reacts.

Google Hacking A technique that uses advanced search criteria to find data that should not be public.

Green Thread Threads that are scheduled by a virtual machine or interpreter. They can not not take advantage of the underlying operating system and hardware thread support. Green threads can not be run across multiple processor cores. A green thread automatically blocks all other threads in the process.

Intrusion Detection A type of program that tries to determine intent of a single packet, or a sequence of them. Its goal is to discover hacking and reconnaissance attempts.

Man in the Middle Attack A method of eavesdropping on, and manipulating communication by rerouting the client and server packets through the attackers machine. This is typically done by poisoning the local network address tables(ARP).

Native Thread Threads that are scheduled by the operating system. It can take advantage of the underlying hardware support for threads.

Nessus A program that tries to discover vulnerabilities of various types of services.

Nikto A program similar to Nessus but is focused on HTTP and HTTPS services.

Nmap A program that is used to detect machines and ports connected to networks.

Open Class A class that can have methods added to it during runtime without subclassing it.

OpenVas A fork of Nessus

Packet Injection A technique that circumvents the operating systems network stack allowing packet header data to be crafted by a tester or attacker.

Packet Sniffer A program that collects raw packets to be saved or displayed.

Penetration Testing A collection of testing techniques used to determine the security of a target.

Proxy A program that mimics the functionality of a server the proxy is hiding. Is used to balance loads among a server cluster or as a security measure that act like a firewall at the application layer. A proxy can also be used by an attacker or tester to change data between the client and server.

Raw Packet Custom crafted network packets used in packet injection and the packets created by the operating systems network stack.

Script Kiddie A derogatory term used for an unskilled cracker that exclusively relies on "off the shelf" software to carry out the attacks.

Security Audit Another name for a penetration test.

Social Engineering An attack technique that relies on gaining information or access from people who use the target.

SQL Injection An attack on network systems that relies on databases. An attacker sends SQL commands via HTTP forms or other methods of taking input to see if the system will return the raw data from the database. Commonly used to steal login information. This technique can also be used to attempt to write to the database and corrupt or change the data in some way that is useful to the attacker.

Structured Query Language (SQL)

Swing The standard Java graphical interface library.

Thread The smallest unit of processing that can be run by an operating system. It is generally used to run 2 or more parts of a single program concurrently.

Bibliography

[1] Andres Andreu. *Professional Pen Testing for Web Applications*. Wrox, 2006.

[2] CERT. Cert statistics. `http://www.cert.org/stats`, 2008.

[3] Joris Evers. T.j. maxx hack exposes consumer data. `http://news.cnet.com/T.J.-Maxx-hack-exposes-consumer-data/2100-1029_3-6151017.html`, 2007.

[4] Free Software Foundation. Frequently asked questions about the gnu licenses. `http://www.gnu.org/licenses/gpl-faq.html`, 2010.

[5] Larry Greenemeier. Hack attack means headaches for tj maxx. `http://www.informationweek.com/news/security/cybercrime/showArticle.jhtml?articleID=197003041`, 2007.

[6] Shane Harris. China's cyber-militia. `http://www.nationaljournal.com/njmagazine/cs_20080531_6948.php`, 2008.

[7] US Health and Human Services. Hipaa administrative simplification statute and rules. `http://www.hhs.gov/ocr/privacy/hipaa/administrative/index.html`.

[8] Andrew Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. Addison Wesley, 2007.

[9] Lavasoft. Spyware statistics. `http://www.lavasoft.com/support/spywareeducationcenter/spyware_statistics.php`.

[10] Elinor Mills. Three men indicted in largest u.s. data breach. `http://news.cnet.com/8301-27080_3-10311336-245.html?tag=mncol`, 2009.

[11] Intrinium Networks. Glba regulations for financial institutions. `https://www.intriniumsecurity.com/resources/white-papers/GLBA%20Regulations%20for%20Financial%20Intitutions.pdf/view`.