



**NAVAL  
POSTGRADUATE  
SCHOOL**

**MONTEREY, CALIFORNIA**

**THESIS**

**AUTOMATION OF CYBER PENETRATION TESTING  
USING THE DETECT, IDENTIFY, PREDICT, REACT  
INTELLIGENCE AUTOMATION MODEL**

by

Kendra Deptula

September 2013

Thesis Advisor:  
Thesis Co-Advisor:

Deborah E. Goshorn  
John McEachen

**Approved for public release; distribution is unlimited**

THIS PAGE INTENTIONALLY LEFT BLANK

<b>REPORT DOCUMENTATION PAGE</b>			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
<b>1. AGENCY USE ONLY (Leave blank)</b>		<b>2. REPORT DATE</b> September 2013	<b>3. REPORT TYPE AND DATES COVERED</b> Master's Thesis	
<b>4. TITLE AND SUBTITLE</b> AUTOMATION OF CYBER PENETRATION TESTING USING THE DETECT, IDENTIFY, PREDICT, REACT INTELLIGENCE AUTOMATION MODEL			<b>5. FUNDING NUMBERS</b>	
<b>6. AUTHOR(S)</b> Kendra Deptula				
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Naval Postgraduate School Monterey, CA 93943-5000			<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>	
<b>9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b> N/A			<b>10. SPONSORING/MONITORING AGENCY REPORT NUMBER</b>	
<b>11. SUPPLEMENTARY NOTES</b> The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.				
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> Approved for public release; distribution is unlimited			<b>12b. DISTRIBUTION CODE</b>	
<b>13. ABSTRACT (maximum 200 words)</b>  The design and implementation of a systems approach to a scalable, standardized automated cyber penetration testing system using the Detect, Identify, Predict, React (DIPR) intelligence automation model and data interoperability standards is the focus of this thesis. The system fuses information from multiple freeware programs that can be thought of as cyber sensors into an interoperable, robust whole in a manner that can tailor itself and learn over time. The groundwork is laid for an enduring system that can adapt to changing systems and vulnerabilities. A barebones proof-of-concept system is implemented and tested using NMap and Ettercap with the proposed DIPR XML file formats as the data intelligence automation standardization mechanism. By implementing this automated cyber penetration system, labor-intensive and costly cyber penetration testing can be simplified by reducing the amount of hand coding and manual testing.				
<b>14. SUBJECT TERMS</b> Automated Penetration Testing, DIPR Artificial Intelligence Model, XML Standardization			<b>15. NUMBER OF PAGES</b> 101	
			<b>16. PRICE CODE</b>	
<b>17. SECURITY CLASSIFICATION OF REPORT</b> Unclassified	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b> Unclassified	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b> Unclassified	<b>20. LIMITATION OF ABSTRACT</b> UU	

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release; distribution is unlimited**

**AUTOMATION OF CYBER PENETRATION TESTING USING THE DETECT,  
IDENTIFY, PREDICT, REACT INTELLIGENCE AUTOMATION MODEL**

Kendra Deptula  
Lieutenant, United States Navy  
B.S., United States Naval Academy, 2006

Submitted in partial fulfillment of the  
requirements for the degree of

**MASTER OF SCIENCE IN ELECTRICAL ENGINEERING**

from the

**NAVAL POSTGRADUATE SCHOOL  
September 2013**

Author: Kendra Deptula

Approved by: Deborah E. Goshorn  
Thesis Advisor

John C. McEachen  
Thesis Co-Advisor

R. Clark Robertson  
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

## **ABSTRACT**

The design and implementation of a systems approach to a scalable, standardized automated cyber penetration testing system using the Detect, Identify, Predict, React (DIPR) intelligence automation model and data interoperability standards is the focus of this thesis. The system fuses information from multiple freeware programs that can be thought of as cyber sensors into an interoperable, robust whole in a manner that can tailor itself and learn over time. The groundwork is laid for an enduring system that can adapt to changing systems and vulnerabilities. A barebones proof-of-concept system is implemented and tested using NMap and Ettercap with the proposed DIPR XML file formats as the data intelligence automation standardization mechanism. By implementing this automated cyber penetration system, labor-intensive and costly cyber penetration testing can be simplified by reducing the amount of hand coding and manual testing.

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS

<b>I.</b>	<b>INTRODUCTION, OVERVIEW AND BACKGROUND .....</b>	<b>1</b>
<b>A.</b>	<b>OVERVIEW .....</b>	<b>1</b>
	1. Thesis Problem Statement, Goals, and Objectives .....	1
	2. Approach to Proposed Solution.....	2
	3. DIPR Intelligence Automation System Architecture.....	2
	4. DIPR Intelligence Automation Data Interoperability Standards.....	5
	5. Implemented Proof-of-concept System.....	6
<b>B.</b>	<b>STAKEHOLDER NEED.....</b>	<b>7</b>
	1. Tactical Need .....	7
	2. Operational Need .....	7
	3. Strategic Need.....	8
	4. Acquisitions Need.....	8
<b>C.</b>	<b>BACKGROUND .....</b>	<b>8</b>
	1. Detect, Identify, Predict, React Intelligence Automation.....	8
	<i>a. Detect Subsystem.....</i>	<i>9</i>
	<i>b. Identify Subsystem .....</i>	<i>9</i>
	<i>c. Predict Subsystem .....</i>	<i>9</i>
	<i>d. React Subsystem.....</i>	<i>9</i>
	2. Cyber Penetration Testing .....	10
	3. Field Experience with Cyber Penetration Operators .....	12
<b>D.</b>	<b>LITERATURE SEARCH .....</b>	<b>14</b>
	1. Penetration Testing.....	14
	2. XML as Middleware.....	14
	3. San Jose State University Automated Penetration System.....	15
	4. HackSim System.....	16
	5. Solar Sword System .....	17
	6. Conclusions from Literature Search.....	18
<b>E.</b>	<b>THESIS STRUCTURE .....</b>	<b>19</b>
<b>F.</b>	<b>SUMMARY .....</b>	<b>19</b>
<b>II.</b>	<b>A SYSTEMS APPROACH TO INTELLIGENCE AUTOMATED CYBER PENETRATION .....</b>	<b>21</b>
<b>A.</b>	<b>CYBER TERRAIN.....</b>	<b>21</b>
	1. Network-based Cyber Terrain.....	22
	2. Host-based Cyber Terrain .....	22
<b>B.</b>	<b>OPERATOR.....</b>	<b>23</b>
	1. Operator Inputs .....	23
	<i>a. Initial Exploit Development.....</i>	<i>23</i>
	<i>b. Assessment Inputs .....</i>	<i>23</i>
	2. Outputs to the Operator .....	24
<b>C.</b>	<b>DIPR AUTOMATED CYBER PENETRATION SYSTEM .....</b>	<b>24</b>
	1. Freeware Programs (Cyber Sensors).....	24

	a.	<i>Discovery Programs</i> .....	25
	b.	<i>Attack Programs</i> .....	25
	c.	<i>Requirements for Freeware Programs</i> .....	25
2.		<b>DIPR Intelligence Automation Interoperability Standards</b> .....	26
	a.	<i>Network Entity XML</i> .....	26
	b.	<i>Exploit XML</i> .....	27
3.		<b>GUI (Command and Control Interface)</b> .....	28
4.		<b>Detect Identify Predict React (DIPR) Modules</b> .....	28
	a.	<i>Detect Module</i> .....	29
	b.	<i>Identify Module</i> .....	30
	c.	<i>Predict Module</i> .....	31
	d.	<i>React Module</i> .....	32
D.		<b>CONCLUSIONS</b> .....	33
	1.	<b>Desire for Stealth</b> .....	33
	2.	<b>Growth and Standardization</b> .....	33
	3.	<b>Summary</b> .....	34
III.		<b>PROOF-OF-CONCEPT</b> .....	35
A.		<b>FREEWARE SELECTION</b> .....	35
	1.	<b>Discovery Programs</b> .....	35
	a.	<i>Network Mapping (NMap)</i> .....	35
	b.	<i>Ettercap</i> .....	36
	c.	<i>Metasploit</i> .....	36
	2.	<b>Attack Programs</b> .....	36
	a.	<i>Ettercap</i> .....	37
	b.	<i>Cain &amp; Able</i> .....	37
	c.	<i>Denial of Service (DoS) Attack</i> .....	37
	3.	<b>Conclusions on Freeware Programs</b> .....	37
B.		<b>NETWORK ARCHITECTURE</b> .....	38
	1.	<b>Topology</b> .....	38
	2.	<b>Software and Computer Specifications</b> .....	38
C.		<b>DIPR INTELLIGENCE AUTOMATION STANDARDS USING XML FILES</b> .....	39
	1.	<b>Exploit XML</b> .....	39
	2.	<b>Network Entity XML</b> .....	41
D.		<b>GUI</b> .....	41
E.		<b>DIPR MODULES</b> .....	42
	1.	<b>Detect Module</b> .....	42
	2.	<b>Identify Module</b> .....	43
	3.	<b>Predict Module</b> .....	44
	4.	<b>React Module</b> .....	45
F.		<b>DETERMINING SUCCESS</b> .....	46
G.		<b>CONCLUSIONS</b> .....	49
	1.	<b>Intelligence Automation Success and Flexibility</b> .....	49
	2.	<b>Stealth</b> .....	49
	3.	<b>Standardization and Scalability</b> .....	49

4.	Summary.....	49
IV.	IMPROVEMENTS, FUTURE WORK AND CONCLUSIONS .....	51
A.	IMPROVEMENTS TO THE PROOF-OF-CONCEPT .....	51
1.	Operating Environment .....	51
2.	Robustness .....	52
B.	FUTURE WORK.....	52
1.	Additional Assessment Module.....	53
2.	Predict Module .....	54
3.	GUI.....	54
4.	Intelligent Automation of New Freeware .....	55
C.	CONCLUSIONS .....	55
1.	DIPR System.....	55
2.	Proof-of-concept.....	55
	APPENDIX A .....	57
	APPENDIX B .....	71
	LIST OF REFERENCES.....	73
	INITIAL DISTRIBUTION LIST .....	75

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF FIGURES

Figure 1.	The DIPR automated cyber penetration system external system.....	3
Figure 2.	The proposed DIPR automated cyber penetration system.....	4
Figure 3.	Assessment scenario flow chart. From [13].....	13
Figure 4.	DIPR automated cyber penetration model external systems. ....	21
Figure 5.	Example network entity XML. ....	26
Figure 6.	Example exploit XML. ....	27
Figure 7.	GUI data flow. ....	28
Figure 8.	DIPR automated cyber penetration system.....	29
Figure 9.	Detect module flow.....	30
Figure 10.	Identify module flow.....	31
Figure 11.	Predict module flow.....	32
Figure 12.	React module flows.....	33
Figure 13.	Network topology. ....	38
Figure 14.	Sample of the exploit XML. ....	40
Figure 15.	Example network entity XML. ....	41
Figure 16.	Screenshot of GUI.....	42
Figure 17.	Network entity XML after the Detect module.....	43
Figure 18.	Network entity XML after the Identify module.....	44
Figure 19.	Network entity XML after the Predict module. ....	45
Figure 20.	Network entity XML after the React module. ....	46
Figure 21.	Wireshark capture of ARP poisoning the network via Ettercap. ....	47
Figure 22.	MiTM attack in the network. ....	48
Figure 23.	Intercepted traffic viewed in Wireshark. ....	48
Figure 24.	DIPR internal diagram with additional Assessment module. ....	53

THIS PAGE INTENTIONALLY LEFT BLANK

**LIST OF TABLES**

Table 1. Comparison of literature review solutions to proposed DIPR solution. ....18  
Table 2. Comparative look at requirements and operating systems in freeware programs. ....52

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF ACRONYMS AND ABBREVIATIONS

DIPR	Detect, Identify, Predict, React
ESD	External Systems Diagram
JIFX	Joint Interagency Field Experimentation
MiTM	Man-in-the-Middle
S&T	Science and Technology
PEO	Program Executive Officer
PMW	Program Management Warfare Office
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance
DBMS	Relational Database Management System
DoS	Denial of Service
IDS	Intrusion Detection System
VM	Virtual Machine
DCO	Defensive Cyber Operations
OPNAV	Office of the Chief of Naval Operations
XML	Extensible Markup Language
JIFX	Joint Interagency Field Experimentation
HTTP	Hyper Text Transfer Protocol
SIP	Session Initiation Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
OS	Operating System
NMap	Network Mapping

THIS PAGE INTENTIONALLY LEFT BLANK

## EXECUTIVE SUMMARY

As industry becomes more reliant on complex computer systems using well-known protocols, it becomes increasingly important to ensure vulnerabilities are caught and corrected early [1]. A method of addressing this need is to focus on the automation of penetration testing in order to reduce time and manpower required to test well-known vulnerabilities. The impact of rising levels of complexity of software and networks creates a dynamic environment requiring additional time and effort to adequately adapt tools and techniques [1].

One approach to detecting vulnerabilities is cyber penetration testing. This task is most beneficial to increasing cyber security; however, there is a high cost of time, money, and manpower associated with manual cyber penetration [1].

With increased computing power available, intelligent automation is a clear choice for simplifying the lives of both administrators and developers. Actual implementation, however, is harder when the long-term scalability and evolution of both vulnerabilities and exploits must be taken into account. Data needed to pass information between automated systems needs to be standardized into an interoperable data schema. In order to scale, standardize, and automate intelligence, a system of systems methodology to automating cyber penetration testing is needed. This is the high-level goal and problem statement for this thesis.

To achieve the high-level goal of automating cyber penetration testing, there are three objectives for this thesis: (1) use a systems approach to design an architecture that automates the tasks of a cyber-penetration operator, (2) design an interoperable data format to interpret the automation systems, and (3) implement a proof-of-concept system of the modeling/design objectives (1) and (2).

A unique systems solution to the high-level goal of automating cyber vulnerability evaluation and penetration of a target network is presented in this thesis. In order to achieve the first objective of systems architecture design for automating a cyber-penetration system, the Detect, Identify, Predict, React (DIPR) intelligence automation

model is utilized. In previous work the DIPR framework has been applied to cyber for determining defensive cyber-attacks, where the detect modules attempt to dynamically detect cyber red team activity [2]–[5]. In [2], the dynamic defensive cyber sensors were Intrusion Detection Systems, and the research focused on the optimal configuration of such systems in order to detect and identify states indicative of red team activity. In contrast to [2], the DIPR framework is used in this thesis to detect network activity using freeware programs as cyber sensors that are needed in order to sense the vulnerability of a network device in the cyber terrain. This systems approach yields a long term solution to automating cyber penetration with the ability for the system to scale by selecting “plug and play” cyber penetration tools. It is a system that maximizes the processing the computer performs to automatically analyze targets, properly identify vulnerabilities, and, finally, perform automated cyber penetration attacks.

The foundation for creating a scalable and implementable automated system that maximizes individual components is to have a clear architecture planned beforehand. By endeavoring to enable any new program to fit and feed into this system, a flexible and long-lasting tool is produced that can adapt as both vulnerabilities as well as the methods for exploiting those vulnerabilities change.

The approach taken by this thesis is to first generate a system of systems architecture for automating cyber penetration testing, which includes both system design and design of the needed data interoperability standards for the data transmitted between each intelligence automation system.

The external systems diagram (ESD) of the proposed cyber systems of systems approach for automating cyber penetration of a network connects directly into the targeted cyber network. The external systems diagram of such a proposed solution is depicted in Figure 1. The external systems are the human operator and the targeted network system needing cyber penetration, denoted as “Cyber Terrain.” The operator in this case initially is deeply involved in developing exploits for specific vulnerabilities to tailor to the DIPR Automated Cyber Penetration System. Subsequent to tool development, the operator is involved to a limited extent while the DIPR system is

executing, allowing the automated system to process and launch the desired exploit. The operator may choose to validate the automated process and provide feedback.

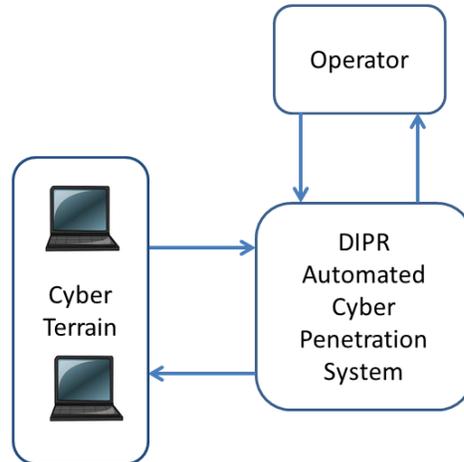


Figure 1. The DIPR automated cyber penetration system external system.

Breaking the proposed system into a more detailed view, we can see the data flow between the modules as depicted in Figure 2. The DIPR Automated Cyber Penetration System includes four intelligence automation modules that transform cyber data from data created when initially sensing the cyber terrain using freeware programs. Then the data is transformed into intelligence automation representations associated with four modules of intelligence automation: Detect, Identify, Predict, and React. Besides the DIPR modules, there are the Graphical User Interface (GUI), network and exploit data structures. The data flow begins with an interface between the DIPR system and the cyber terrain which captures the data using various freeware programs (Metasploit, NMap, Xprobe2, etc.). In order for the solution to scale and adapt to the disparate nature of the outputs of the various freeware programs, a translation step must occur. Thus, the proposed system's functionality includes the ability to transform these outputs into useable, interoperable middleware formats.

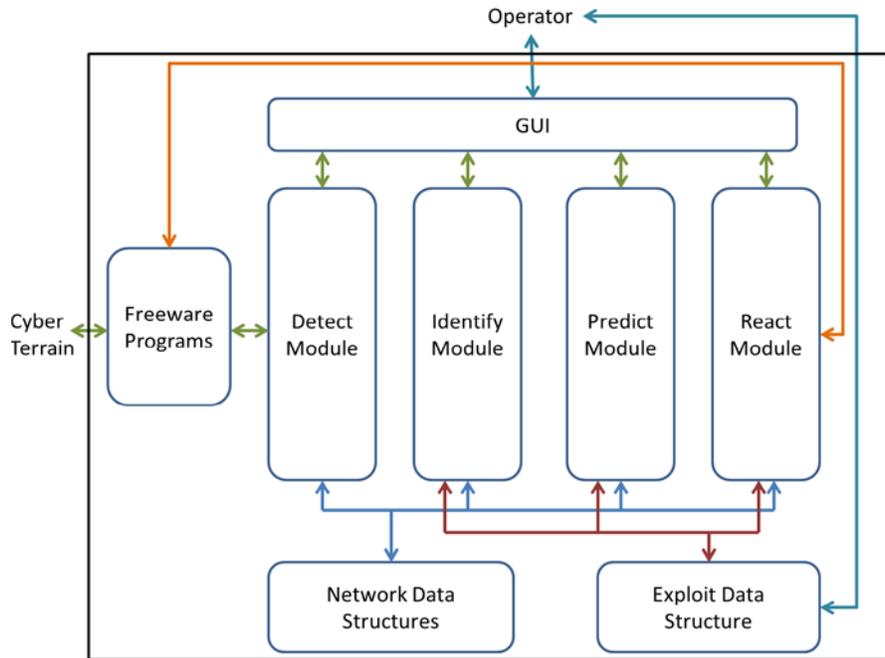


Figure 2. The proposed DIPR automated cyber penetration system.

The graphical user interface serves as the command and control (C2) structure for the system. As each module performs its specific automated task, the GUI automatically allows the next module to begin. This is also where the operator has the option to manually select and specify exactly which tasks to accomplish against the chosen target.

For the sake of complexity management and storage reduction, only those features that are useful to exploits are retained. The Detect module creates a feature database per network entity, where a feature of a cyber-entity is considered a technical detail or piece of information about that host or network (open port, operating system, etc.) which later can be used to analyze that entity. Detect features are represented and stored in data structures as part of the proposed DIPR Intelligence Automation Interoperability standards for data representation. Similarly, the details required for each exploit are stored in their own data structure. These data structures are maintained in a tree formatted extensible markup language (XML) document.

Each of these features is then processed within the Identify module to determine vulnerabilities, where an intelligent state can be identified based on a group of specific features occurring simultaneously. If vulnerabilities exist, then that host's data structure

is updated accordingly. Several fused features (i.e., intelligent states) occurring in a pattern within a period of time result in them being classified as a behavior in the predict module.

After the state sequences are formed and vulnerability behaviors are classified, the predict module processes the inferred vulnerability behavior outcome by determining which exploits are available against the target. Based on operator input and likelihood of success, a recommended cyber penetration attack is generated and added to the vulnerability data structure.

When the inferred recommendation has been made, the react module uses all of the accumulated data to then launch the penetration attack most likely to succeed against the target computer. Upon completion, it determines if the attack was successful and feeds that information back into the exploit data structure to update the expected probability of success for that attack against that vulnerability.

The DIPR intelligence automation interoperability standards are used to represent data generated by the DIPR modules. Specific XML schemas which map to the DIPR modules in order to represent data between them are proposed in this thesis. By implementing output/input of DIPR modules using XML files, the DIPR Automated Cyber Penetration system can scale with more freeware programs (cyber sensors) and more discovery/react tools.

Following the generated architecture, we implement the architecture in a proof-of-concept system, including both software implementations within a virtual network along with data interoperability standards implementation using proposed structured file formats.

The final proof-of-concept network was set up using a simple client server configuration. Using freeware together with the backbone architecture, a man-in-the-middle (MiTM) attack was launched which poisoned the network and subsequently intercepted the data flowing between the client and the webserver. For each success, the program updated the exploit database probability of success for that penetration attack exploiting that vulnerability. The starting status for the penetration attack machine was a

blind entrance onto the network with all target IP addresses unknown. There was one server, one penetration attack box (running a basic version of the architecture proposed in this thesis), and a client which sent information back and forth across the network to a web server. Overall success was determined by the positive integration of freeware programs for network discovery and attack as well as successful vulnerability analysis and attack prediction.

Ultimately, this system does not remove the need for an experienced analyst evaluating vulnerabilities or methods of exploiting these vulnerabilities. It helps provide a new long-lasting tool that will reduce the time needed to manually test and exploit a network. As operating systems and network traffic change, so will the need for tools used to properly identify key vulnerabilities in an exploited system, underscoring the importance of being able to add new tools as fluidly and rapidly as possible to the system. This will benefit organizations in operational, tactical, strategic and acquisitions positions that have the need to determine, prioritize, and address network vulnerabilities of networks quickly and efficiently.

## LIST OF REFERENCES

- [1] E. Chow, "Ethical hacking & penetration testing," University of Waterloo, Waterloo, Canada, 2011.
- [2] B. Jurjonas, "Smart selection and configuration of cyber sensors for active defensive cyber operations," M.S.. thesis, Department of Electrical and Computer Engineering, Naval Postgraduate School, March 2012.
- [3] "Network operations," class notes for CY 3602, Department of Electrical and Computer Engineering, Naval Postgraduate School, Winter 2012.
- [4] D. Goshorn, "Cyber security and cyber Warfare in a Network-Centric Smart Sensor System of Systems," brief given to RADM William Leigher. Naval Postgraduate School, Monterey, CA, 2010.
- [5] D. Goshorn, R. Goshorn. "Cyber security in a network-centric smart-environment system of systems" presented at the Cyber Summit Conference. Monterey, California, 2009.

THIS PAGE INTENTIONALLY LEFT BLANK

# **I. INTRODUCTION, OVERVIEW AND BACKGROUND**

An overview of the problem statement, goals and an approach to solving the problem are presented in this thesis. Following the overview is a breakdown of stakeholder needs, a literature review of various automated cyber penetration systems, and a summary of the remaining three chapters.

## **A. OVERVIEW**

The goals and objectives of this research as described in the problem statement, and the proposed three-tiered approach to solving the problem are provided in this section.

### **1. Thesis Problem Statement, Goals, and Objectives**

As industry becomes more reliant on complex computer systems using well-known protocols, it is increasingly important to ensure vulnerabilities are caught and corrected early on [1]. A method of addressing this need is to focus on the automation of penetration testing in order to reduce the time and manpower required to test well known vulnerabilities. The impact of rising levels of complexity of software and networks creates a dynamic environment requiring additional time and effort to adequately adapt tools and techniques [1].

One approach to detecting vulnerabilities is cyber penetration testing as described in a subsequent section of this chapter. This task is most beneficial to increasing cyber security; however, there is a high cost in time, money, and manpower associated with manual cyber penetration [1].

With increased computing power available, intelligent automation is a clear choice for simplifying the lives of both administrators and developers. Actual implementation is harder when the long term scalability and evolution of both vulnerabilities and exploits must be taken into account. Data needed to pass information between automated systems needs to be standardized into an interoperable data schema. To scale, standardize, and automate intelligence, a system of systems methodology to

automating cyber penetration testing is needed. This is the high-level goal and problem statement for this thesis.

To achieve the high-level goal of automating cyber penetration testing, there are three objectives for this thesis: (1) use a systems approach to design an architecture that automates the tasks of a cyber-penetration operator, (2) design an interoperable data storage format to facilitate interpretation automation systems, and (3) implement a proof-of-concept system of the modeling/design objectives (1) and (2).

## **2. Approach to Proposed Solution**

A unique systems solution to the high-level goal of automating cyber vulnerability evaluation and penetration of a target network is presented in this thesis. In order to achieve the first objective of systems architecture design for automating a cyber-penetration system, the Detect, Identify, Predict, React (DIPR) intelligence automation model is utilized. In previous work the DIPR framework has been applied to cyber for determining defensive cyber-attacks, where the detect modules attempt to dynamically detect cyber red team activity [2]–[5]. In [2], the dynamic defensive cyber sensors were Intrusion Detection Systems, and the research focused on the optimal configuration of such systems in order to detect and identify states indicative of red team activity. In contrast to [2], the DIPR framework is used in this thesis to detect network activity using freeware programs as cyber sensors that are needed in order to sense the vulnerability of a network device in the cyber terrain. This systems approach yields a long term solution to automating cyber penetration with the ability for the system to scale by selecting “plug and play” cyber penetration tools. It is a system that maximizes the processing the computer performs to automatically analyze targets, properly identify vulnerabilities, and, finally, perform automated cyber penetration attacks.

## **3. DIPR Intelligence Automation System Architecture**

The foundation for creating a scalable and implementable automated system that maximizes individual components is to have a clear architecture planned beforehand. By endeavoring to enable any new program to fit and feed into this system a flexible and

long lasting tool is produced that can adapt as both vulnerabilities as well as the methods for exploiting those vulnerabilities change.

The approach taken in this thesis is to first generate a system of systems architecture for automating cyber penetration testing, which includes both system design and design of the needed interoperability standards for the data transmitted between each intelligence automation system. The former is discussed in this section.

The external systems diagram (ESD) of such a proposed solution is depicted in Figure 1. The external systems are the human operator and the targeted network system to be penetrated, denoted as “Cyber Terrain.” The operator in this case initially is deeply involved in developing exploits for specific vulnerabilities to tailor to the DIPR Automated Cyber Penetration System. Subsequent to tool development, the operator is involved to a limited extent while the DIPR system is executing, allowing the automated system to process and launch the desired exploit. The operator may choose to validate the automated process and provide feedback.

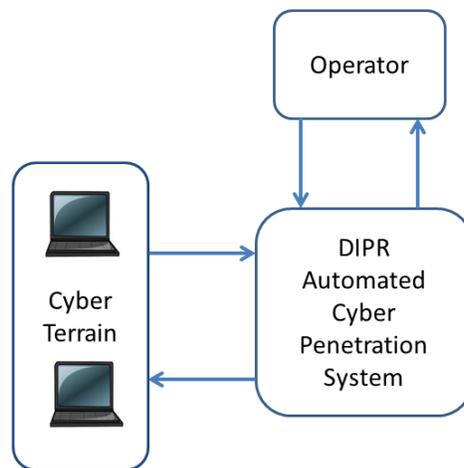


Figure 1. The DIPR automated cyber penetration system external system.

Breaking the proposed system into a more detailed view, it is possible to see the data flows between the modules as depicted in Figure 2. The DIPR Automated Cyber Penetration System includes four intelligence automation modules that transform cyber data from data created when initially sensing the cyber terrain using freeware programs.

Data is then transformed into intelligence automation representations associated with the four modules of intelligence automation Detect, Identify, Predict, and React (DIPR), which are explained in the subsequent section. Besides the DIPR modules, there are the Graphical User Interface (GUI), network and exploit data structures. The data flow begins with an interface between the DIPR system and the cyber terrain that captures the data using various freeware programs (Metasploit, NMap, Xprobe2, etc.). In order for the solution to scale and adapt to the disparate nature of the outputs of the various freeware programs, a translation step must occur. Thus, the proposed system's functionality includes the ability to transform these outputs into useable, interoperable middleware formats. Such data structure formats are discussed in the subsequent section.

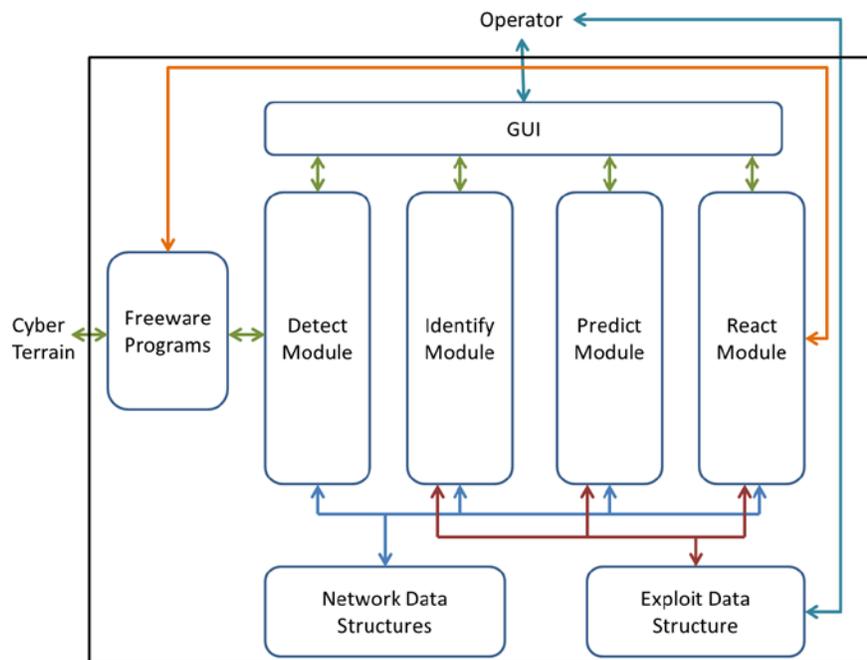


Figure 2. The proposed DIPR automated cyber penetration system.

The graphical user interface serves as the command and control (C2) structure for the system. As each module performs its specific automated task, the GUI automatically allows the next module to begin. This is also where the operator has the option to manually select and specify exactly which tasks to accomplish against the chosen target.

For the sake of complexity management and storage reduction, only those features which are useful to exploits are retained. The Detect module creates a feature database per network entity, where a feature of a cyber-entity is considered a technical detail or piece of information about that host or network (open port, operating system, etc.) which later can be used to analyze that entity. Detect features are represented and stored in data structures as part of the proposed DIPR Intelligence Automation Interoperability standards for data representation. Similarly, the details required for each exploit are stored in their own data structure. These data structures are maintained in a tree formatted Extensible Markup Language (XML) document.

Each of these features is then processed within the Identify module to determine vulnerabilities, where an intelligent state can be identified based on a group of specific features occurring simultaneously. If vulnerabilities exist, then that host's data structure is updated accordingly. Several fused features (i.e., intelligent states) occurring in a pattern within a period of time will result in then being classified as a behavior in the Predict module.

After the state sequences are formed and vulnerability behaviors are classified, the predict module processes the inferred vulnerability behavior outcome by determining which exploits are available against the target. Based on operator input and likelihood of success, a recommended cyber penetration attack is generated and added to the vulnerability data structure.

When the inferred recommendation has been made, the react module uses the accumulated data to launch the penetration attack most likely to succeed against the target computer. Upon completion, it determines if the attack was successful and feeds that information back into the exploit data structure to update the expected probability of success for that attack against that vulnerability.

#### **4. DIPR Intelligence Automation Data Interoperability Standards**

The approach taken in this thesis is to generate a system of systems architecture for automating cyber penetration testing, which includes both system design and design

of data interoperability standards needed for the data transmitted between each intelligence module. An overview of the latter is discussed in this section.

The DIPR intelligence automation interoperability standards are used to represent data generated by the DIPR modules. Specific XML schemas are proposed which map to the DIPR modules in order to represent data between them. By implementing output/input of DIPR modules using XML files, the DIPR Automated Cyber Penetration system can scale with more freeware programs (cyber sensors) and more discovery/react tools.

## **5. Implemented Proof-of-concept System**

Following the generated architecture, we implement the architecture in a proof-of-concept system, including software implementations within a virtual network along with data interoperability standards implemented using the proposed structured file formats. An overview of the proof-of-concept system is provided in this section.

The final proof-of-concept network was set up with a simple client server configuration. Using freeware together with the backbone architecture, the system sets up a man-in-the-middle (MiTM) attack, poisoning the network and subsequently intercepting the data flowing between the target computer and the webserver. For each success the system updates the exploit database probability of success for that penetration attack exploiting that vulnerability. The starting status for the penetration attack machine was a blind entrance onto the network with all target IP addresses unknown. There was one server, one penetration attack box (running a basic version of the architecture proposed in this thesis), and a client, which sent information back and forth across the network to a web server. Overall success is determined by the positive integration of freeware programs for network discovery and attack as well as successful vulnerability analysis and attack prediction.

Additionally, the GUI and XML schemas and file formats used are described in depth in Chapter III.

Ultimately, this system does not remove the need for an experienced analyst evaluating vulnerabilities or methods of exploiting these vulnerabilities. It provides a new long lasting tool which reduces the time needed to manually test and exploit a network. As operating systems and network traffic change, so will the need for tools used to properly identify key vulnerabilities in an exploited system, underscoring the importance of being able to add new tools as fluidly and rapidly as possible to the program. This will benefit organizations in operational, tactical, strategic and acquisitions positions which have the need to determine, prioritize, and address network vulnerabilities of networks quickly and efficiently.

## **B. STAKEHOLDER NEED**

The need for this thesis is categorized into the following perspectives: tactical, operational, strategic, and acquisition.

### **1. Tactical Need**

From a tactical perspective, the Navy Red team deals primarily with gaining unauthorized access, testing for vulnerabilities and determining weaknesses of Navy networks with the intent of exposing problems for correction [6]. The Red team will benefit from this study by having an intelligent automated system reduce time spent manually gaining access and probing for weaknesses. From the vulnerabilities exposed, the Navy Blue team, responsible for maintaining and hardening Navy networks against malicious penetration, can then endeavor to address and harden those vulnerabilities, making the likelihood of successful penetration attack from an outside force less likely [6].

### **2. Operational Need**

From an operational perspective, Tenth Fleet/Fleet Cyber Command, the Navy component of U.S. Cyber Command, is responsible for information security, computers and cryptologic concerns of the entire Navy, both at sea and on shore [7]. Using the information gathered by the Red Team, fleet-wide changes can then be implemented that ensure common vulnerabilities are not exploited. These operational changes will have a large impact on fleet security.

### **3. Strategic Need**

From a strategic perspective, two organizations will benefit. Firstly, the newly merged OPNAV N2 (Intelligence)/ N6 (Communications), which is a sister group to Tenth Fleet, is responsible for future planning for fleet cyber defense [8]. They will be able to take a long term look at current issues to create durable solutions to ensure that the navy Cyber community is not solely reactionary. Knowledge of such an automated system will provide OPNAV N2/N6 the opportunity to steer related aspects of the science and technology (S&T) of this area in cyber by demonstrating the ability to quickly determine system vulnerabilities as an important component of future networks as a watchdog for emerging cyber vulnerabilities. Secondly, USCYBERCOM will benefit from this tool as an aid in strategic cyber mission planning across all services [7].

### **4. Acquisitions Need**

From an acquisitions perspective, the Program Executive Office Command and Control, Communications, Computers, and Intelligence (PEO C4I and PMW 130), dedicated to providing products dedicated to information security and cyber defense for the fleet, along with other PMWs in PEO C4I, will be aided from the acquisitions approach to cyber systems in the Navy's C4ISR systems [9].

## **C. BACKGROUND**

General background on intelligence automation, penetration testing, and field experience with penetration operators is provided in this section.

### **1. Detect, Identify, Predict, React Intelligence Automation**

The intelligence automation model used in this thesis implements the DIPR automation model and has been used primarily in sensor based networks to generate useable, intelligent feedback from raw data provided by conventional sensors such as cameras [11]. It has been used once before for automating dynamic defensive cyber operations (DCO), which focused primarily on the cyber sensors and detection portions [2].

**a. *Detect Subsystem***

Starting with a standard system there are basic sensors in the field feeding raw data back to a processing center running the DIPR program. From here the raw sensor data is analyzed, and raw features are created in a Detect Subsystem. As previously discussed, features are a low level classification created from raw data [11].

**b. *Identify Subsystem***

From this point, the Identify Subsystem processes the features, recognizes when multiple features have occurred simultaneously and fuses them together, generating a state to which that data belongs. Rules to implement the recognition must be developed ahead of time to determine what features are required but, ideally, it would be an adaptable learning system which could tailor itself over time [11].

**c. *Predict Subsystem***

The state generated by the Identify Subsystem is then input into the Predict Subsystem. These states are then mapped and generalized to create high-level classifiers which use the inputs of states, location and time to develop sequences or behaviors. They are then classified as “normal” behaviors, “abnormal” behaviors and “unclassified” behaviors based on predefined or learned patterns. These behaviors are then used to predict the future state of the system under observation and output to the React Subsystem [11].

**d. *React Subsystem***

Once handed to the React Subsystem, these predicted behaviors are acted upon. The output of the React Subsystem is an action appropriate to the predicted behavior (warnings, alarms, etc.) [11].

In general, the DIPR system is well-suited for application to the cyber field. In Chapter II, more specific adaptations are addressed as to how the sensors and terrain are described and mapped to work in this model.

## **2. Cyber Penetration Testing**

Cyber Penetration testing is generally the action of a system administrator or software engineer deliberately searching for vulnerabilities and determining what information can be determined about their system. Manually performing this process is extremely time consuming and, generally, only results in determining that those specific tests were unable to find any vulnerabilities. To ease the burden of this process, many suites and programs have been developed to automatically perform a subset of some of those tasks, but in certain cases can these programs be linked together to form a more robust system.

Penetration testing both during the software development process as well as at the final production point is crucial to ensuring an end product with limited vulnerabilities. A concern in particular for the software development industry is the trade-off between manual and automatic penetration testing. While manual testing is perhaps more thorough, it is vastly more time and processing power intensive, and basic automation has the limitation of only finding the specific vulnerabilities which it is programmed to find. Penetration testing in general is limited in its efficacy due to the fact that it does not prove that a system does not have vulnerabilities, simply that it does not have any of the vulnerabilities that were specifically searched for. In many cases this might be sufficient for the first release with any subsequent problems being addressed in patches. To this end, several freeware programs have been developed in order to minimize the amount of time taken to manually penetration test a network. In this thesis we focus on well-known freeware programs. All have the ability to both scan and attack, and both aspects are employed in the final integration of the system.

The specific people who are interested in using the DIPR Automated Cyber Penetration System likely come from a fairly specialized niche of hackers. The term “hacker” in the context of this thesis refers to someone who can gain access to a computer network without using the standard methods defined by the organization’s policy. Hackers have a wide variety of motivations for accessing a system and what they will do if successful. As new technology and software enters the market, new flaws and

security vulnerabilities similarly emerge. The market for finding and fixing these flaws has grown in proportion to how prevalent technology has become.

The first type of person who would be intent on exploiting these flaws is a “black hat” hacker. Black hatters enter into a system without authorization and steal information or damage the system in some way [12]. They can be anyone from a script kiddie to an elite corporate hacker looking for secrets. The tools they use could range from freeware to personally written and scripted code tailored to enter into a system [12]. Black hatters are what most organizations fear and, therefore, expend considerable resources ensuring that they are either adequately protected or repairing damage after realizing that they were not.

The second type, on the opposite side of the spectrum, would be the “white hat” or “ethical hacker.” They are typically hired by a company to look for flaws in the system and determine what vulnerabilities exist [12]. They use any tools at their disposal to attempt to break into the system the same way a black hatter does so that they may find as many flaws as possible and help the company correct them or guard against them [12]. This system in particular can help them use a wide variety of freeware tools to quickly, in various combinations, launch attacks and determine which have the highest chances of success. Because they are being paid by the company for results, it is most profitable to execute as many attacks and avenues as rapidly as possible. However, this brings up one of the primary flaws in cyber security: the entire process is one of exclusion. A tool or person can prove that the method that they used does not work, but it is impossible to prove that there exists no method which could work. It is a fundamental flaw and is one of the reasons that well planned security from the initial development of every program or network is essential. Automated tools will not fix this flaw but can greatly increase the number of common attacks launched using the most common tools directed against a network in various combinations.

The third type falls somewhere in between the two extremes and, aptly, is called a “grey hat” hacker. These are typically people that illegally enter a network and find flaws in the system but, instead of exploiting or damaging the network, inform the company of the issue so that they may correct it [12]. Motivations for this type of person

are varied and unpredictable. This system could also be useful for grey hatters with the added benefit of not leaving any additional footprints outside of what is normal for freeware programs being used manually. For instance, if a hacker were to specifically tailor code to break into a bank's computer network, they leave a specific signature behind that may link their code to them and any other instances of that code. However, with an automated system utilizing only freeware programs, it is more difficult to discern a specific person manually running the programs while still providing the added benefit of decreased time.

### **3. Field Experience with Cyber Penetration Operators**

During the 2012 Joint Interagency Field Experimentation (JIFX) held at Camp Roberts in February, the author met with several of the Navy cyber red team members. These people were tasked with finding vulnerabilities of specific projects using typical freeware programs available to the general public. In some cases they knew well in advance what program they would be acting against and, in others, they were told with little or no time to prepare. When given sufficient lead time for research, they employed a flow chart (as seen in Figure 3) or checklist to find as much information about the system beforehand as possible. This resulted in consistently documentable and traceable results which lent itself to better corrections by the programs security team. They expressed an interest in automated tools to easily combine freeware programs and to help more quickly move step by step through the process. Throughout the exercise they typically used more than six different programs to find and utilize information about the target system and to determine the most successful avenue of approach. From this initial assessment, they would use a series of other freeware programs to launch attacks. In the case of new programs where little prior knowledge was available, this process was conducted employing either trial and error or brute force methods.

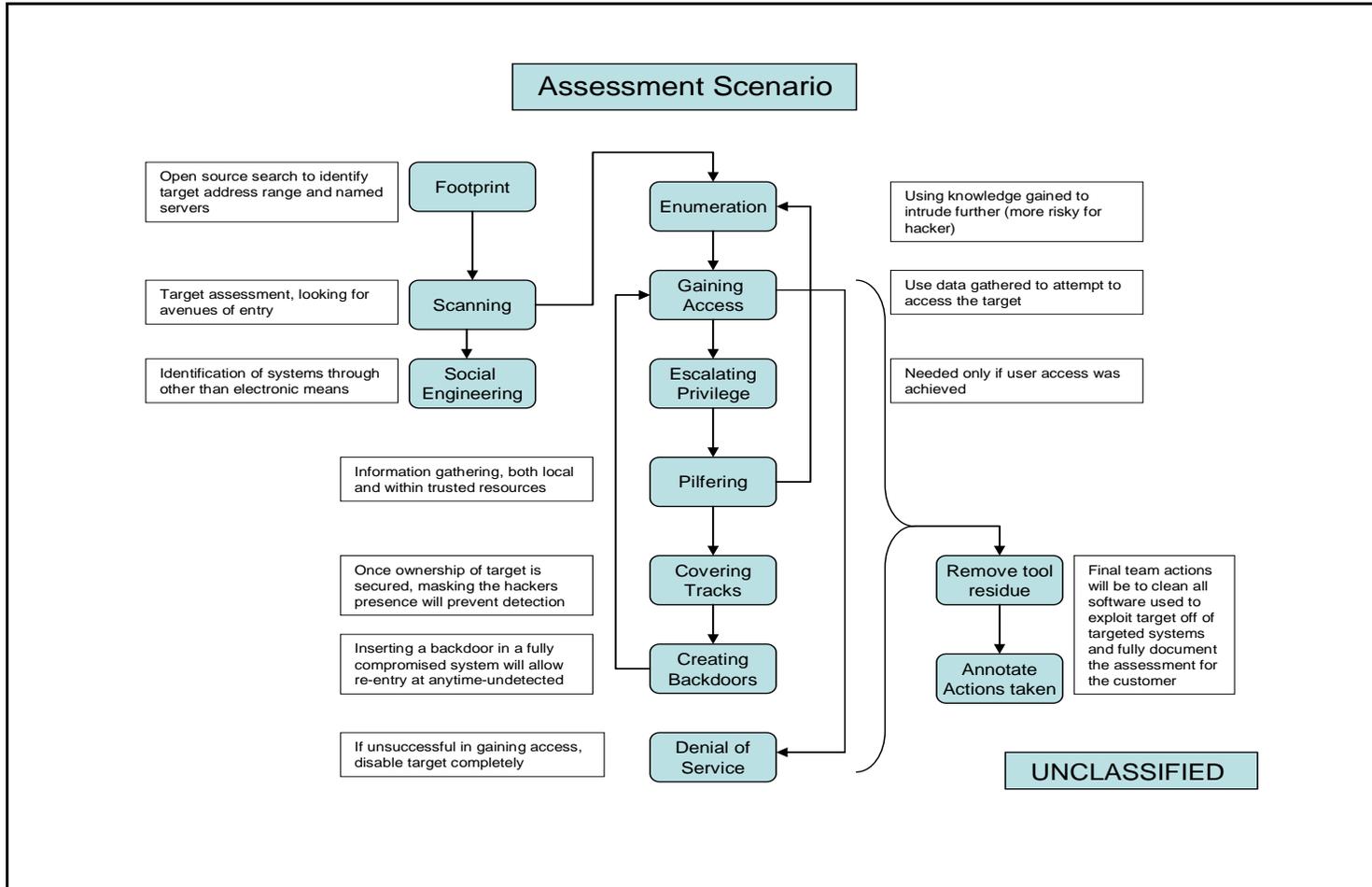


Figure 3. Assessment scenario flow chart. From [13].

Ultimately, this system will benefit the Navy by assisting people in red team roles by quickly assessing and determining weaknesses in new programs prior to them being pushed into the Fleet.

## **D. LITERATURE SEARCH**

Several other efforts to automate cyber penetration are described and middleware implementation recommendations are made in this section.

### **1. Penetration Testing**

The paper by J.P. McDermott, “Attack Net Penetration Testing,” discusses various methodologies of finding, approaching, and eliminating programming flaws [10]. In the case of this thesis, the intent is only to find and exploit flaws, but the process remains the same. The “flaw hypothesis approach” breaks down the process into a series of steps: 1. define goals, 2. background research, 3. generate hypothetical vulnerabilities, 4. confirm vulnerabilities, 5. generalize flaws, 6. eliminate flaws [10]. For the purpose of this thesis, we begin at step six of this approach, maximize the flaw and then exploit the flaw. McDermott’s view of using an attack net for a MiTM attack is consistent with the structure of this thesis; although, he pursued a different end goal. The concept of an attack net is each previous step must be completed to unlock access to the following step, which in turn allows access to the next step and so forth to the goal. This is modeled by portraying intermediate and final objectives as places or nodes on a map, commands and inputs as transitions between these nodes, and using “tokens” as placeholders for the current position in the map [10]. In an ideal situation, it is possible to either find a simpler path to the goal or make it possible that, on a second attack, the path is left open for further exploitation using this methodology.

### **2. XML as Middleware**

Choosing an efficient, scalable method for storing information is the focus of the article “XML Data Stores: Emerging Practices” [14]. Because of XML’s ease of use, interoperability and reliability, it has increasingly become the data store of choice for the programming industry [14]. The implementation of these XML data stores can vary from

application to application, and there are a variety of benefits to the various methods. Options discussed were: ASCII files stored in a management system as objects, relational database management system (DBMS) (Tables), object relational DBMS (tables and objects), native XML, and directory servers [14]. Given the various pros and cons of the preceding options, using a native XML has the qualities of scalability, reliability and data access speed. Native XML is defined as a data store that uses XML as its fundamental storage unit, uses a logical model for the XML itself, and requires a specific physical storage method [14]. Information can then be stored either in a tree format or a collection format. Both are valid methods and best practice is determined by the needs of the application. A tree stores the data in a parent child configuration, whereas the collection method groups them in larger sets using either a typed schema [6] or as untyped unrelated XML documents [14]. The use of a Native XML inherent to the application as well as a tree storage method was the method utilized in this thesis.

### **3. San Jose State University Automated Penetration System**

The thesis Automated Penetration Testing by Neha Samant examines the need for automated penetration testing to reduce the time and cost of manually performing the penetration testing [15]. By performing penetration testing early and often in either the network development or software development stages, it is easier and less time-consuming to identify or correct issues as opposed to waiting until the system is finalized [15]. The paper developed a web-based penetration system to create a variety of user friendly denial of service (DoS) attacks. The application was successful in accomplishing basic DoS attacks using three protocols: hypertext transfer protocol (HTTP), session initiation protocol (SIP) and transmission control protocol/Internet protocol (TCP/IP). For each protocol two to twenty attacks were available. The user friendly interface takes the input of the IP address to be attacked and a port number to direct that attack. The application successfully completed these attacks with a user friendly interface that simplified penetration testing as opposed to performing the attacks manually.

Weaknesses in this method are limited scalability in the future and the continuing need to use external programs to gather information prior to attack. With no previously established architecture, the application development is constrained and possibly limited in efficiency. The lack of integration with a port or IP discovery program reduces the scope and degree of intelligent automation of the system. The focus of this thesis is on creating an architecture that can be put in place prior to development of the web application to ensure long term scalability as well as integrating programs to fully automate the system.

#### **4. HackSim System**

Another implementation of penetration testing called HackSim was created to automate penetration testing for remote buffer overflow vulnerabilities [16]. By developing an architecture that focused on discovery and exploit (similar to this thesis's discovery and attack programs), they created a modular approach which focused on a user-friendly interface, automation and safety for the target program. Designed to work either in a white box (tester has a priori knowledge of the system) or black box (tester has no a priori knowledge of the system) environment testing, the intent was to sanitize code used for penetration, ensuring that while the system was tested it would not be damaged and no backdoors were inadvertently left in place. This allowed the system or network to be tested while still in operation, a situation which frequently occurs in the commercial world after a new program or function has been installed on the network. The system was successfully implemented against Solaris and Windows systems.

Hacksim does have the ability to be expanded and scaled to larger implementations but has to be manually coded [16]. This limits how advanced and time relevant this framework would be long term. In this specific framework, there was little reason for standardization of the implementation of either the discovery or exploit modules. However, this could result in difficulties in a more advanced implementation.

In addition to similarly working to automate penetration testing, the intent of this thesis is to focus on implementing the intelligence between the discovery and exploit steps.

## **5. Solar Sword System**

A third implementation, named Solar Sword, is the closest to the scope of this thesis [17]. Using an automated, modular system they developed a distributed network of secure information gathering platforms to feedback information to a central command and control center [17]. The three focus areas for this implementation are automation, distribution and immunity from external attacks. The approach uses the distributed program on a LiveDVD system to run programs to determine information about the network and send that information back to the central station [17]. The central station processes the information, determines the best course of attack based on possible vulnerabilities and generates a template for attack which is disseminated to the distributed testing clients. After retrieval the clients launch the attacks and generate reports about the success of the attack. These are then sent back to the central control station for manual analysis [17].

This method uses a standardized approach to its attack template as well as utilizing the information sent back from the distributed clients, which provided more information from a wider range of vantage points. This allows for error checking gathered data and for potentially harvesting more information depending on distributed client vantage point [17]. Additionally, its use of automation and intelligence to parse the received information and process it through either an attack net or attack graph allows it to adapt to a previously unknown network and determine the best course of testing. The implementation was successful in achieving its goals and future work was determined to be necessary in the areas of increasing the automation and intelligence in its attack graph and securing the communications between the central control center and its distributed clients [17]. Finally, they concluded that further work needed to be done to make the implementation more robust. In Solar Sword, an approach to designing a methodology which was also scalable and easily upgradeable to new exploits is important and achievable but is not a primary focus for Solar Sword [17].

A different approach is taken in this thesis by emphasizing modularity in order to change to a rapidly developing environment while focusing on using premade freeware programs to do the majority of the work for the penetration tester. The use of the freeware programs frees much of the burden of relying on native scripts and exploits to test the system.

## 6. Conclusions from Literature Search

Based on the above literature search, the focus of this thesis is on establishing a DIPR based architecture using a methodical approach to flaw recognition and exploitation. Using XML as the data store format, we integrate disparate freeware programs into a scalable system which allows future expansion and reliability.

A comparison of each reviewed implementation as compared to each of our focus areas is illustrated in Table 1. Primarily, these areas are scalability, standardization and intelligence. In the context of this thesis, scalability refers to the ability of the implementation to easily adapt to additional growth of the network or of desired exploits. This can typically be seen by a modular architecture but is not limited to this. Manually coding or developing each vulnerability can also significantly limit the scalability due to the time and cost associated with manual coding and detecting. To a certain extent, avoiding manual coding is impossible, but steps such as standardization and intelligence can reduce this work. Standardization refers to making any inputs to modules used or information gained to classify vulnerabilities as having similar classifications for their attributes. Intelligence refers to the ability to mesh several details or vulnerabilities in order to recognize a new and different vulnerability. This can also refer to the ability to self-recognize or adapt to given inputs.

Table 1. Comparison of literature review solutions to proposed DIPR solution.

	<b>Scalability</b>	<b>Standardization</b>	<b>Intelligence</b>
San Jose State University Implementation		X	
HackSim	X		
Solar Sword		X	X
Proposed DIPR Implementation	X	X	X

Focus areas lacking from the other implementations is the concept of intelligently automating the system to allow it to recognize multiple aspects and identify them as a

more advanced vulnerability. Standardization was not necessary in all of the implementations but was not discussed as a key point either. Each implementation had a unique approach, and all were successful in their stated goals but a more structured, well defined architecture can assist in the automation process.

#### **E. THESIS STRUCTURE**

The stakeholder need driving this thesis was articulated in Chapter I followed by the general background required for this thesis. A literature search on other efforts in automating penetration testing concluded the chapter. The recommended architecture of the DIPR Automated Cyber Penetration System are described in Chapter II. The software implementation for the proof-of-concept system of the proposed DIPR Automated Cyber Penetration System and discussion of the results of that implementation are described in Chapter III. Finally, recommended improvements to the proof-of-concept, future work and conclusions are provided in Chapter IV.

#### **F. SUMMARY**

A general overview of the need to develop a scalable, flexible architecture for the automation of network penetration and exploitation was presented in this chapter. Detail on the overall thesis structure, stakeholder needs, background behind the chosen architecture and a summary of additional literature in this area was provided. Further details on the development of this architecture are addressed in Chapter II.

THIS PAGE INTENTIONALLY LEFT BLANK

## II. A SYSTEMS APPROACH TO INTELLIGENCE AUTOMATED CYBER PENETRATION

The proposed methodology for automatic penetration testing and exploitation management within a cyber system of systems framework is the focus of this chapter. The systems framework is depicted in Figure 4. There are two external systems to the DIPR Automated Cyber Penetration System: the cyber terrain that is to be penetrated and the operator. The external systems, the DIPR Automated Cyber Penetration System, and the information flows between each system are described in detail in the following sections.

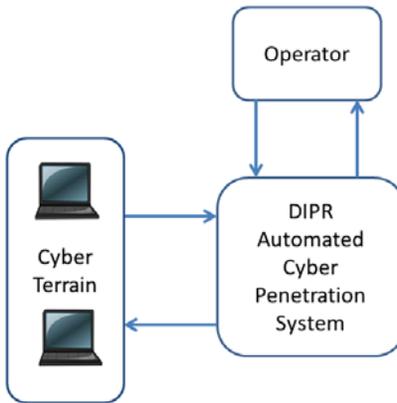


Figure 4. DIPR automated cyber penetration model external systems.

### A. CYBER TERRAIN

With most cyber scenarios, the primary issue is to clearly delineate where the terrain of the problem begins and ends. In this thesis that boundary is set at the outputs of our sensors (freeware programs which are discussed more in depth further on) which feed into our system. This clearly establishes that the sensors themselves are not integrated into the system (with the exception of input commands) but instead only the outputs must be manipulated to conform. This ensures that we prevent creating a fingerprint, allowing an outside observer to realize the system is automated vice a person manually performing the operations. The assumption is that the attack computer has a physical connection to

the target network. For this thesis, cyber terrain is broken into two separate categories: network-based terrain and host-based terrain [3]. Both are ideally utilized, but only the network-based terrain is featured in the proof-of-concept.

### **1. Network-based Cyber Terrain**

Network-based cyber terrain can be considered any physical connection to a device on the network (node) as well as each network connection combination between any two nodes on the network (edges) [3]. To gather information in a network-based cyber terrain, our system sends out data in order to process and receive data to determine features or behaviors of the target system. Examples of network terrain include computers, routers, storage systems and the physical connections between them which make up the network itself. Knowledge about the terrain is limited to that which can be gathered by our sensors.

### **2. Host-based Cyber Terrain**

Host-based cyber terrain differs from network-based by how the information is obtained by our system. In a host-based cyber terrain, the target computer, with the implementation of a previously installed back door, processes and determines features and behaviors itself and send that information to our system. Data gathered by the host-based terrain can be considered all the information regarding one device on the network (the internals of the device) which does not normally get transmitted between two nodes on the network [3]. Examples of information in a host-based cyber terrain include the system logs, the number of processes running, and the CPU utilization on a given computer.

Unlike the network-based cyber terrain, the host-based requires an advanced implementation where a program or malware created by one of our freeware programs has left a backdoor in place to later retrieve information. Here it is assumed that the target computer is sending information and establishing a connection without being prompted by the attack computer.

## **B. OPERATOR**

Automation of a penetration system, regardless of how advanced or well implemented, never completely alleviates the need for a human analyst to help find new vulnerabilities. The system is only able to automate at the level to which it is programmed. Additionally, in order to ensure that the correct target is chosen and acted against, it is always wise to ensure that a human component can be inserted into the process. In this thesis, the operator has several roles as described subsequently.

### **1. Operator Inputs**

The involvement of the operator for this architecture is discussed in this section. Both the expectations for initial exploit development as well as the expected received inputs and outputs are explained.

#### *a. Initial Exploit Development*

Research into system vulnerabilities is the first step in the process for exploiting a system or network. After a program or system is launched, the most likely source of bugs or flaws is the end-users. Extensive time and development is directed towards debugging programs before they are released into the market, but invariably, there are a few issues that still crop up. Those users who find the issues may provide this information to the company, exploit the flaws themselves, or just post the information on various blogs or message boards. As a result, it is valuable for a tester to regularly peruse these boards and look for new exploits or flaws which may further provide access to a system. In this thesis it is expected that an operator or analyst populates a database ahead of time with possible vulnerabilities with as many details as possible about a system that the vulnerability applies to and what the expected result is for those actions. The validity of this database is paramount in executing a successful automated attack.

#### *b. Assessment Inputs*

Determining and tracking the success or failure of an exploit on a system with a particular vulnerability is essential to correctly choosing which exploits to utilize in future cases. For this thesis this information is provided by the user. Further

development on automating methods of effectiveness aids in removing this input, but some human interaction should be involved to monitor the quality of the attack.

## **2. Outputs to the Operator**

Primarily, the outputs for the system are the status of what step of the process the system is performing so that the user can choose to monitor outputs. This is to provide situational awareness of the status of the automated system to the operator. Additional monitoring of the network is expected through a program such as Wireshark or tcpdump to view the effectiveness of a network based attack.

## **C. DIPR AUTOMATED CYBER PENETRATION SYSTEM**

A general description of how each component is structured, followed by a more in-depth look at the data flow within each module and from one module to another, is discussed in this section.

### **1. Freeware Programs (Cyber Sensors)**

The DIPR architecture has primarily been implemented in the automation of standard physical sensors feeding back to artificial intelligence software. In this system we are moving from the physical environment to the cyber environment. As such, the definitions of each component change slightly; although, the intended goal remains the same.

The DIPR cyber sensors are considered to be the probes sent from the programs used by the penetration software. They represent the interface that our attack computer has with the target network. Just as there are two types of cyber terrain, there are two categories of cyber sensors. Those determining information about network based information and those that interact with implants on network computers (to extract information about the host) which can send back information. For this thesis, host based sensors are not implemented but should be mentioned as another crucial sensor in a fully fleshed out implementation. Network based sensor data can be thought of as the data being sent from the attack computer to elicit a response in the target computer. For

example, packet captures from TCP or ICMP probes are sensor data from the network-based cyber sensors NMAP and XProbe2, respectively.

Additionally, further delineation must be made between the actual roles of each program. Penetration testing can be seen as two separate functions. The first function is a discovery function, responsible for exfiltrating information about the network. Second is an attack function which is responsible for disrupting, intercepting or denying some service. Programs used are associated with these functions for use in this system, and it is possible that a single freeware program can serve in either role. A more thorough discussion of specific programs is addressed in Chapter III.

***a. Discovery Programs***

Discovery programs can also later double in the React module as attack programs depending on the robustness of the freeware program. In general, a discovery program is any program that can elicit information from the network. This can also be either honed down in the event that the target IP address is known (i.e., instead of scanning the entire network it specifically targets a single computer) or bypassed entirely if enough information is already known on the target.

***b. Attack Programs***

Attack programs are any type of freeware that take inputs and then launch a specific attack against the network.

***c. Requirements for Freeware Programs***

To make an architecture which is scalable, limitations on what the system can or cannot accept in its freeware program has to be balanced with ensuring that the full range of programs are utilized. Current network toolboxes such as Metasploit are readily available and easy to use. Developing Metasploit plugins that allow toolboxes to interact with this architecture is necessary but should be done cautiously such that the final output does not look automated when processed through our system.

Because of the requirement to have systems that interact directly with our system, any discovery program used must have a command line interface available and output that can be piped or parsed such that it can be converted into a standardized XML format. Any attack program used only needs to have command line input; although, a parseable output is useful for determining success.

## 2. DIPR Intelligence Automation Interoperability Standards

Because XML is both ubiquitous as well as extremely versatile from both a human readable and machine readable perspective, it has been chosen for this thesis as the primary method of data storage in all applications. The standardization of XML fields allows for synchronization of data between disparate programs, which increases the information available about a particular network while reducing the redundant information taking up storage.

### a. Network Entity XML

The XML file created which holds all data pertaining to a specific entity on the network (i.e., computer, router, storage device, connection, etc.) is referred to as a network entity XML. An example is provided in Figure 5.

```
<?xml version="1.0"?>
<Address>
  192.168.120.150
  <TimeStamp>2013-06-21 09:28 Coordinated Universal Time</TimeStamp>
  <ProgramUsed>Nmap 6.25</ProgramUsed>
  <OpenPort>22</OpenPort>
  <OpenPort>80</OpenPort>
  <OpenPort>139</OpenPort>
  <OpenPort>443</OpenPort>
  <OpenPort>445</OpenPort>
  <OpenPort>515</OpenPort>
  <OpenPort>548</OpenPort>
  <OpenPort>873</OpenPort>
  <OpenPort>3689</OpenPort>
  <OpenPort>8873</OpenPort>
  <OpenPort>9050</OpenPort>
  <OpenPort>22939</OpenPort>
  <MacAddress>00:24:A5:59:2E:86</MacAddress>
  <OSGuess>Linux 2.6.X</OSGuess>
</Address>
```

Figure 5. Example network entity XML.

*b. Exploit XML*

Prior to beginning the DIPR process, an exploit database must be created associating vulnerabilities with possible exploits that can take advantage of these vulnerabilities. For this thesis this database is stored as an XML file and referred to as the exploit XML. An example exploit XML is provided in Figure 6.

```
<?xml version="1.0"?>
- <Exploits>
  - <Vulnerability>
    A
    - <Aspect>
      OS
      <Spec>Linux 2.6.X</Spec>
    </Aspect>
    - <Aspect>
      Port
      <Spec>80</Spec>
    </Aspect>
    - <Recommend>
      MiTM-Ettercap
      <Prob>5:8</Prob>
      <Requires>B</Requires>
    </Recommend>
    - <Recommend>
      MoTS-Cain and Able
      <Prob>3:7</Prob>
    </Recommend>
  </Vulnerability>
  - <Vulnerability>
    B
    - <Aspect>
      OS
      <Spec>Microsoft Windows 2000|XP</Spec>
      <Spec>Windows XP SP2</Spec>
    </Aspect>
    - <Aspect>
      Port
      <Spec>3389</Spec>
      <Spec>6667</Spec>
      <Spec>7000</Spec>
    </Aspect>
    - <Recommend>
      MiTM-Ettercap
      <Prob>5:10</Prob>
    </Recommend>
    - <Recommend>
      DOS-Cain and Able
      <Prob>6:10</Prob>
    </Recommend>
  </Vulnerability>
</Exploits>
```

Figure 6. Example exploit XML.

### 3. GUI (Command and Control Interface)

A robust GUI is necessary in order to ensure that all of the options available to the individual freeware programs are still accessible in the integrated system. Additionally, this is where most of the hand over and command and control occur. Having a GUI which still allows the operator to be inserted at any point in the process is also desirable both from a verification standpoint as well as to ensure no actions are taken without understanding the consequences. The data flow for the GUI is shown in Figure 7. It interfaces directly with the four modules and the operator. To maintain modularity, the GUI should have no connections to external programs.

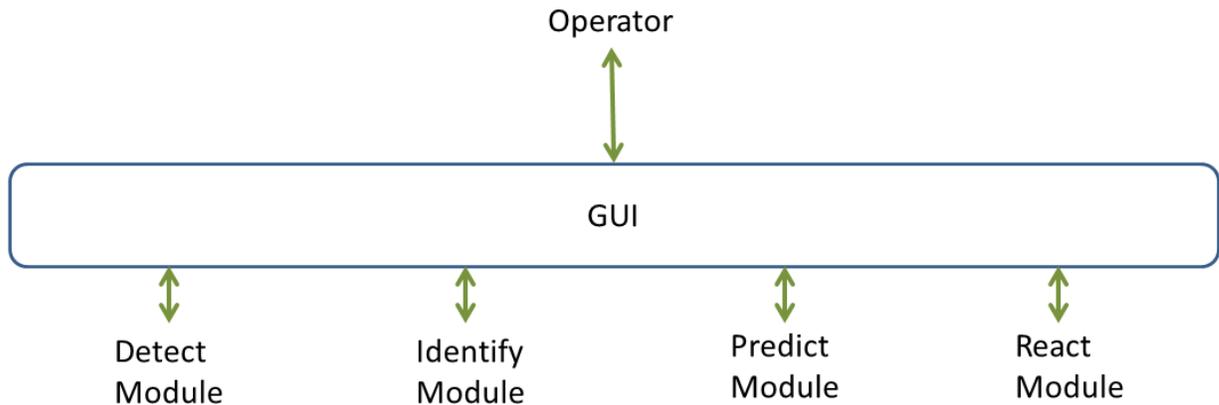


Figure 7. GUI data flow.

### 4. Detect Identify Predict React (DIPR) Modules

The tasks assigned to each module and the separation of control for each module is the focus of this section. An overall flow diagram for the modules as a whole is provided in Figure 8, and the flows for the individual modules are shown in subsequent sections.

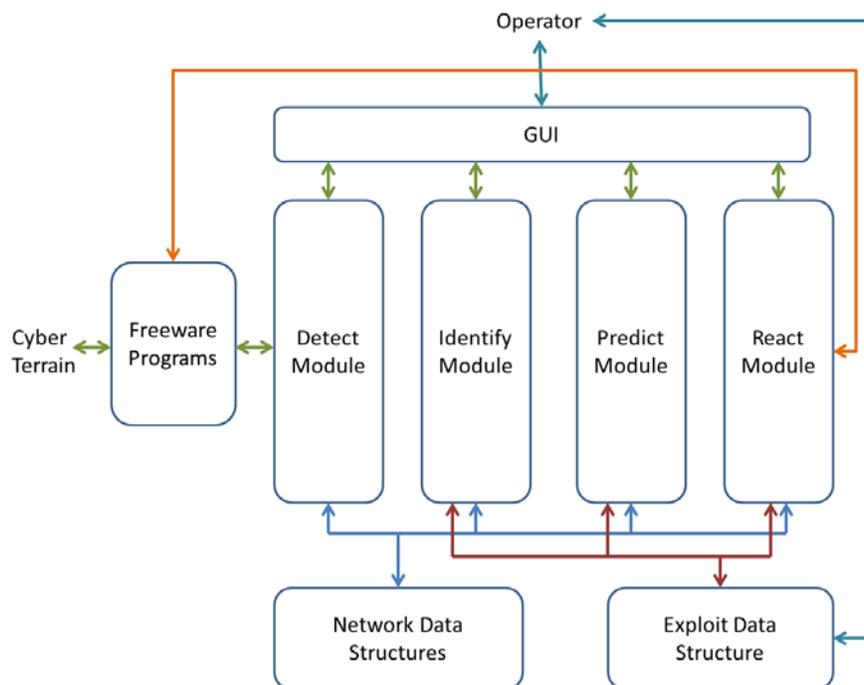


Figure 8. DIPR automated cyber penetration system.

*a. Detect Module*

The detect module has four primary tasks:

- Run the discovery program
- Parse the output of the discovery program.
- Create an XML document per network entity
- Make/update a master network entity list

As shown in Figure 7, the flow begins with the GUI calling on the Detect module with either the name of the specific requested discovery program or with no input, allowing the module to use the default. Once the program is run, the output is parsed to gather only the necessary features. Necessary features are determined by the vulnerabilities stated in the exploit XML. This parsed information is then used to create a standardized XML file for each network entry with additional information of MAC address, timestamp and program used to gather the information. Finally, the Detect module creates a master list of all network entities for faster reference in follow on modules. When complete, the module returns a signal to the GUI that it has finished and the next module can take over.

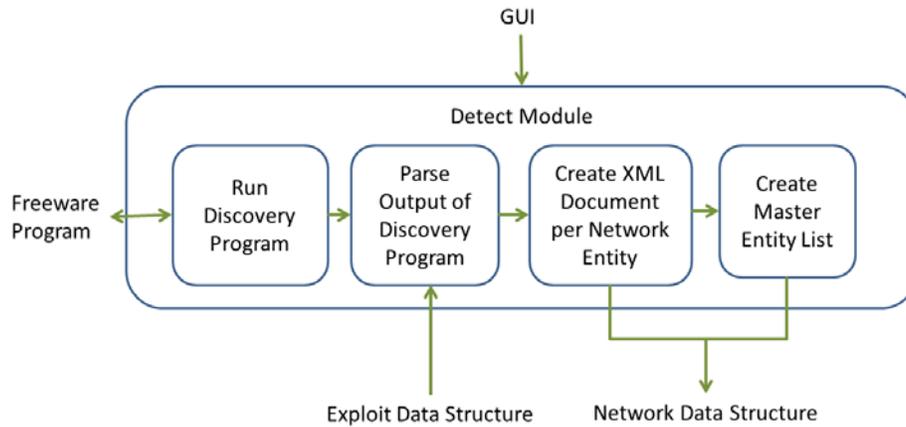


Figure 9. Detect module flow.

**b. Identify Module**

The Identify module similarly has multiple primary tasks:

- Pull features needed for a state from exploit XML file (setting up the query)
- Open each network XML and test to see if the query is true
- Update network XML to indicate that a state exists and what vulnerability it corresponds to.

When the GUI calls on the Identify module, it first looks to the exploit XML file to determine what features are needed to be combined in order to make a state. This can also be seen as building the queries. Then it applies those queries to each of the network entities. For instance if “a” and “b” features are needed for state 0 with a vulnerability “c” (this is the query) and the network entity has both “a” and “b”, then the XML is updated that that entity is in state 0 with vulnerability “c.” State 0 is used to denote that the entity has a vulnerability that can be exploited. A State 1 or higher indicator can occur in the event that a backdoor has been left on the machine from a previous engagement. The difference between these two is what allows for further determination of behavior patterns for that entity. Finally, it returns to the GUI that it has completed successfully. This flow can be seen in Figure 10.

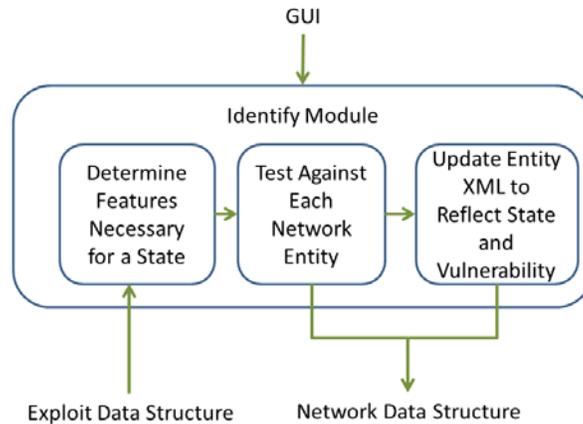


Figure 10. Identify module flow.

*c. Predict Module*

The tasks for the Predict module are:

- Find all possible exploits for each vulnerability
- Check each network entity for the existence of State 0
- Calculate the probability for each type of exploit available for that vulnerability
- Determine if additional information or vulnerabilities are required for that type of attack
- Update network entity XML with recommendation and requirements for the exploit with the highest probability

In Figure 11, we can see the flow of information in the Predict module.

When triggered by the GUI, the Predict module takes the exploit XML file and reads in the possible exploits for each type of vulnerability. Then it calculates the probability of success for each type of attack. Probability can be dependent on different variables such as whether the desired response is the most likely to succeed, produce the desired outcome (MiTM or DoS) or least likely to be detected. The Predict module should have the ability to discern which exploit fits the request best. The module should then look

through all the entity XMLs and find which have a State 0 and update them with the recommended exploits for that vulnerability and any additional requirements. Finally, it should inform the GUI that it has completed its tasks.

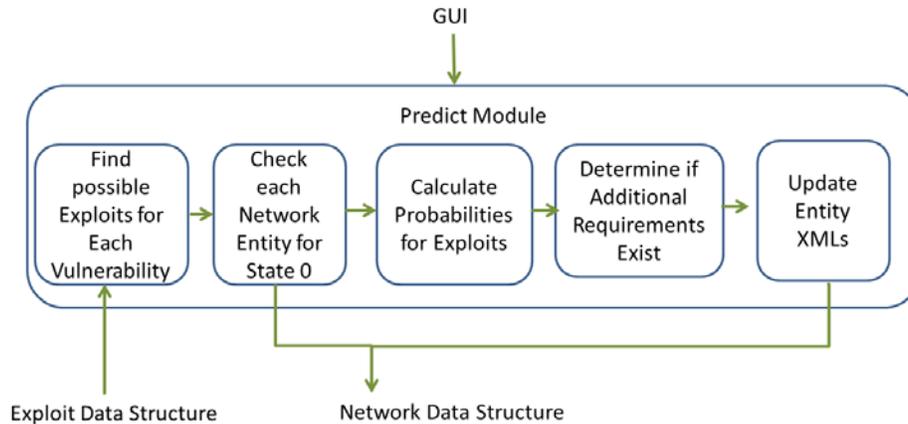


Figure 11. Predict module flow.

#### *d. React Module*

The tasks for the React module are:

- Inspect all network entity XMLs and find the exploit with the highest probability.
- If additional requirements are needed, find them.
- Launch attack using all available information.
- Update the network entity XML with a timestamp of the attack.
- Update the exploit XML with Success or failure.

In Figure 10, the flows for the React module are shown. It begins with the GUI starting the process, triggering the module to look at all of the entity XMLs to determine the exploit with the highest probability of attack. The returned exploit may then have another requirement (another vulnerability or specific piece of information). In that case the module checks to see if that additional condition exists. If it does then it launches the chosen attack using the selected attack program. If it does not then it repeats the search process and chooses the exploit with the next highest probability of success. After the attack is launched, it updates the entity XML and the exploit XML to reflect the success or failure of the attack. This data flow is shown by Figure 12.

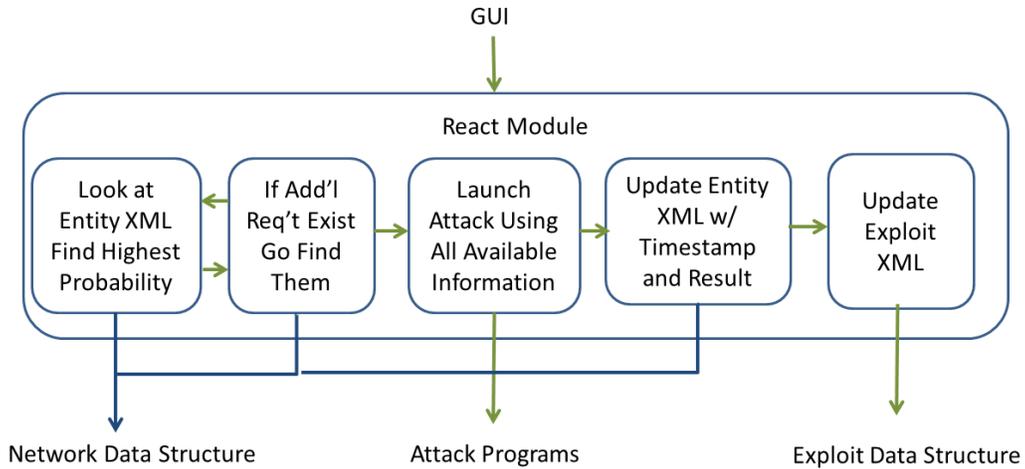


Figure 12. React module flows.

## D. CONCLUSIONS

Conclusions on the systems approach to intelligence automated cyber penetration testing were provided and two important points that were considered while making the architectural choices to utilize the DIPR process specifically and freeware programs in general are identified in this section.

### 1. Desire for Stealth

One of the frustrations when trying to reverse engineer an attack is determining who sent the code that caused the damage to your system. One method a grey hatter can use to avoid identification or finger-printing of themselves is to only use freeware or publicly owned code. This has two consequences: ease of use which allows the operator to spend less time specifically tailoring code and “recreating the wheel” for a particular exploit and, more importantly, it makes it marginally harder to connect that particular version of the code with the specific operator.

### 2. Growth and Standardization

Plug and play programming driven by the use of XML is a focal point for determining a recommended method for formatting and standardizing the outputs of individual programs. Because each new program on the market is unlikely to use a

similar output method, there needs to be a standardized process that allows easy integration with new freeware programs. Determination of specific events and features are stored and expandable as more utilities are used and advanced vulnerabilities are identified. Ease of access and a small footprint assists in maximizing the computational speed of the program and allows for faster real time execution in determining the best exploit for deployment.

### **3. Summary**

An overall methodology for implementing the DIPR system was proposed and the needs of the specific components were addressed in this chapter. The expected data flows externally between the terrain, operator and the DIPR system and the internal data flows between each of these sections were discussed. Proof-of-concept considerations and implementations are further addressed in Chapter III.

### III. PROOF-OF-CONCEPT

The overall creation and development of the proof-of-concept are discussed in this chapter. The methods used to choose freeware, determine the network the proof-of-concept was built upon and how each module was designed to work as an overall whole as well the results of each step of the proof-of-concept are covered.

#### A. FREEWARE SELECTION

The process for selecting the freeware programs that are used in the development of the proof-of-concept are discussed in this section.

##### 1. Discovery Programs

A discovery program is any program that can elicit information from the network. This can be either honed down in the event that the target IP address is known (i.e., instead of scanning the entire network it specifically targets a single computer) or bypassed entirely if enough information is already known about the target. Requirements for these programs are that they have command line input as well as a form of parseable output. Possible choices are discussed in the following sections. There are many others that are available, but these present a general list.

###### a. *Network Mapping (NMap)*

NMap is a small freeware program which uses erroneous TCP packets with various flags set to attempt to fingerprint the operating system (OS) of the targeted computer. The OS determination is made based on the types of replies the target system returns from each erroneous packet. From these replies, an estimate is made as to what OS is most likely running on the target system [18]. When the version of the target system is known, we have achieved the first step in determining what potential vulnerabilities might be available. A limitation of NMap is that it is relatively “noisy”, and the erroneous packets can likely be noticed by an intrusion detection system (IDS) or system administrator. Possible methods of avoiding this are to provide a dummy IP address or to use a less obtrusive program. In the case of white hat attacks, this is

unnecessary, but in a situation where stealth is required, this could lead to detection. Additionally, NMap can be used for basic network mapping, which is important when the target IP address is not known from the beginning.

NMAP was specifically used by the proof-of-concept as the discovery program.

***b. Ettercap***

Ettercap has the ability to actively and passively gather information using a MiTM attack [19] but typically outputs less information while setting up an attack and so is not considered ideal as a discovery program. It was, however, used as a final attack program.

***c. Metasploit***

Another well-known example of both a discovery as well as an attack program is Metasploit [20]. Metasploit has a well-developed interface with many services to both test for vulnerabilities in addition to exploiting the vulnerabilities once found. The program was created in 2003 by HD Moore and has grown in recent years to be one of the most hardy scanning and exploitation tools on the market [20]. Metasploit streamlines many of the day-to-day penetration steps and allows them to be viewed in a flow like GUI format. Many of the basic steps can be understood and followed by a basic user, but more advanced tools must be fully understood to achieve the full potential of the program. The intent behind this thesis is not to replicate a less robust version of this program but to propose an architecture such that any program may interface with the primary program without needing to specifically become an add on to the main program. This allows for the integration of any new tool that is on the market regardless of the code's source.

**2. Attack Programs**

Attack programs are required to take an input and launch a specific attack against the target network or computer. Depending on the desire of the operator, they can have a wide variety of results.

*a. Ettercap*

Ettercap is a simple freeware program that is supported in Linux and Windows up to XP. It is primarily used to launch a MiTM attack using ARP poisoning and ICMP poisoning [19]. Its inputs can be virtually any combination of IP address, MAC Address and ports; that allows it to be easily tailored to specific needs. Its output does not provide much parseable information, but in conjunction with tcpdump or Wireshark, intercepted packets can be captured and reviewed. For the proof-of-concept, Ettercap was used primarily for its simplicity in command line inputs as well as the variety of input combinations available.

*b. Cain & Able*

Cain & Able is a freeware program primarily used to gather passwords and map networks. Its methodology ranges from brute force or dictionary attacks to more advanced decoding and descrambling. Its most recent version can also be used for ARP poisoning a network and MiTM attacks [21]. Cain & Able was not used in the proof-of-concept only because in the Windows environment it has limited command line input. Using an implementation of the architecture in Linux would make the Cain and Able another valid program to consider integrating.

*c. Denial of Service (DoS) Attack*

While not implemented in the proof-of-concept, a DoS attack is one of the most simple to implement and difficult to defend against attacks. It can be implemented using numerous methods, but the most common is to flood a system with some form of packet and overwhelm the server or even the network itself [22]. One end goal for this attack is to clog the network to the point where services are inoperable or time latent.

**3. Conclusions on Freeware Programs**

For both their simplicity as well as their robustness, NMap and Ettercap were implemented in this proof-of-concept. Overall success or failure needs to be verified via Wireshark or tcpdump and input by the operator, but the validity of the architecture holds even with this limitation.

## B. NETWORK ARCHITECTURE

A detailed description of the network designed to test the proof-of-concept is given in the following section. This includes a description of topology, software and computer specifications.

### 1. Topology

The network was designed in a virtual environment and consisted of three virtual machines (VM) and a network data storage device. A visual of the network topology is provided in Figure 13.

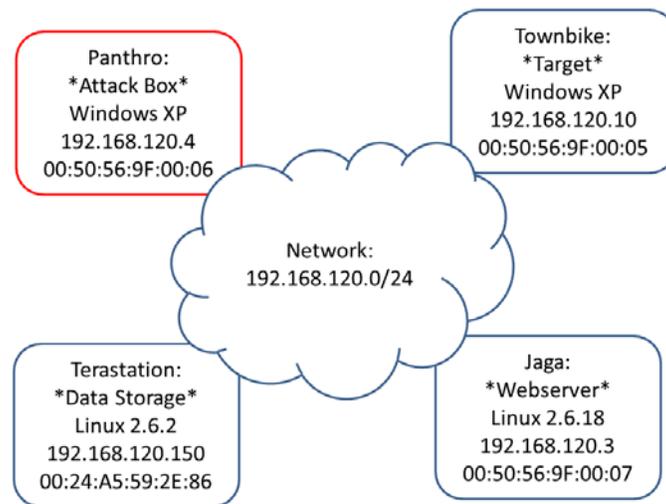


Figure 13. Network topology.

### 2. Software and Computer Specifications

The “Attack Box,” or the machine running the proof-of-concept system, was named Panthro and runs Windows XP with the necessary freeware programs and python installed.

The “Target” machine was similarly running Windows XP with only Wireshark and Internet Explorer installed. Internet Explorer was used to generate traffic to ensure the MiTM attack was operating correctly and Wireshark for troubleshooting and to inspect the packets at the receiving end.

The “Webserver” machine ran an Apache webserver and hosted the webpage that the target computer accesses.

The terastation was the only non-virtual component of the network and was used to back up the virtual machines.

Python was chosen to write all of the code for the proof-of-concept because it is a high level programming language with the ability to process command line input while handling parsing of text files. It also has inherent toolboxes for both GUI development as well as XML manipulation.

### **C. DIPR INTELLIGENCE AUTOMATION STANDARDS USING XML FILES**

The components and details that went into developing the standardization used to create the exploit and network entity XMLs are discussed in this section. Standardization is essential in allowing multiple disparate DIPR programs interface efficiently so that they may be processed by the architecture.

#### **1. Exploit XML**

An example of vulnerability in the exploits XML document is depicted in Figure 14. The vulnerability field names the vulnerability in this example “A”. This is the primary method for labeling the various vulnerabilities that could be found. The naming convention can be substituted for a more complex or descriptive method but increased complexity makes it more likely for error while integrating several programs seamlessly. The aspect field determines the general attribute that is being searched for (i.e., operating system, open port, IP address, etc.) and should be directly related to information that can be provided by a discovery program. The spec field refers to the specific value that should be held by the aspect field. Multiple spec and aspect fields are possible for any vulnerability. Each aspect requires at least one spec value be true for the vulnerability to be considered true. Alternately, it is an “and” operator for aspects and an “or” operator for specs.

Vulnerability A (shown in Figure 14) specifically looks for a Linux OS with an open port 80 (typically this indicates a web server). Two exploits are available to use against this vulnerability: A MiTM attack using Ettercap and a MoTS attack using Cain & Able. These were chosen as examples, and any combination of integrated programs could have been employed. As Cain & Able was not integrated into the system, the probability is initially set lower than that of the MiTM attack to ensure it was not selected. To achieve the MiTM attack, another computer connecting to the webserver is required so the required field indicates the need for vulnerability B.

Vulnerability B (not shown but included in Appendix B) requires that the computer have one of three open ports and be operating Windows XP. In this case the three ports selected were known to be open on our target computer, but if a particular application was known to use a particular port, then this is where we could specifically target that application. Alternately, for this vulnerability a denial of service attack is recommended, which would prevent that computer from timely access to the network.

Together these two vulnerabilities allow a MiTM attack to be established between the client and the webserver.

```

<?xml version="1.0"?>
- <Exploits>
  - <Vulnerability>
    A
    - <Aspect>
      OSGuess
      <Spec>Linux 2.6.X</Spec>
    </Aspect>
    - <Aspect>
      OpenPort
      <Spec>80</Spec>
    </Aspect>
    - <Recommend>
      MiTM-Ettercap
      <Prob>5:8</Prob>
      <Requires>B</Requires>
    </Recommend>
    - <Recommend>
      MoTS-Cain and Able
      <Prob>3:7</Prob>
    </Recommend>
  </Vulnerability>

```

Figure 14. Sample of the exploit XML.

## 2. Network Entity XML

Figure 15 is one example of a network entity XML taken after only being processed by the Detect module. Network entities in this case consider each unique IP address on the network; although, they are not limited by that and the address field indicates that this entity is based upon IP address. A timestamp field is used to give a reference to when the scan was created so that multiple scans can be deconflicted and changes observed. Because various discovery programs have different strengths and weaknesses, the program which was used is included in its own field. Next, all information is gathered and parsed into its individual fields. The field to which it is parsed should match the field being looked for in the exploit XML. As discussed in Chapter II, the Open Ports and OSGuess are called features.

```
<?xml version="1.0"?>
<Address>
  192.168.120.150
  <TimeStamp>2013-06-21 09:28 Coordinated Universal Time</TimeStamp>
  <ProgramUsed>Nmap 6.25</ProgramUsed>
  <OpenPort>22</OpenPort>
  <OpenPort>80</OpenPort>
  <OpenPort>139</OpenPort>
  <OpenPort>443</OpenPort>
  <OpenPort>445</OpenPort>
  <OpenPort>515</OpenPort>
  <OpenPort>548</OpenPort>
  <OpenPort>873</OpenPort>
  <OpenPort>3689</OpenPort>
  <OpenPort>8873</OpenPort>
  <OpenPort>9050</OpenPort>
  <OpenPort>22939</OpenPort>
  <MacAddress>00:24:A5:59:2E:86</MacAddress>
  <OSGuess>Linux 2.6.X</OSGuess>
</Address>
```

Figure 15. Example network entity XML.

## D. GUI

An example screen shot of the GUI is provided in Figure 16 and is a basic indicator of what actions the system is performing. It also provides buttons to indicate failure or success of the final attack in order to update the exploit XML document.

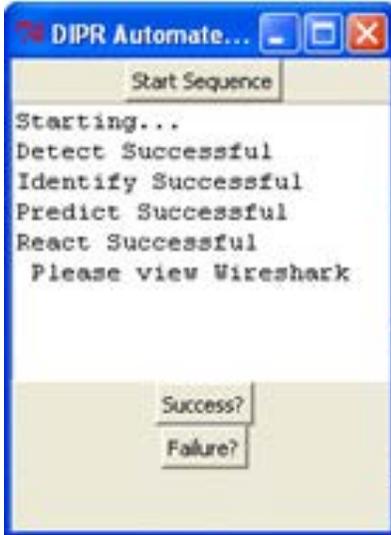


Figure 16. Screenshot of GUI.

## E. DIPR MODULES

The actions of each module and the end result of those actions are stepped through in this section. The general tasks completed are outlined in Chapter II and are discussed in greater depth in the following sections.

### 1. Detect Module

The Detect module is where the discovery program is initially launched and its outputs parsed into the standardized format used by the subsequent modules. The chosen discovery program was NMap, so the first thing the module performs is a system call to NMap using the command:

```
nmap.exe -o 192.168.120.0/24
```

This command is the command to scan all addresses in the network, determine open and closed ports, speculate an OS and record the MAC address of each active IP address on the network. This is sent to a text file which is later parsed.

The module then finds the desired features and parses the specific data from the text file. This is built into the standardized network entity XML for that specific IP Address that those features belong to. This is done for each IP Address that NMap was able to recognize on the network.

The network entity file of the target system after the Detect module has finished parsing the data is shown in Figure 17. The 192.168.120.3 address corresponds to the Jaga webserver in our network, and both the MAC address and OS have been correctly determined.

```
<?xml version="1.0"?>
- <Address>
  192.168.120.3
  <TimeStamp>2013-06-21 09:28 Coordinated Universal Time</TimeStamp>
  <ProgramUsed>Nmap 6.25</ProgramUsed>
  <OpenPort>22</OpenPort>
  <OpenPort>80</OpenPort>
  <OpenPort>111</OpenPort>
  <OpenPort>5432</OpenPort>
  <OpenPort>6667</OpenPort>
  <OpenPort>9080</OpenPort>
  <OpenPort>9618</OpenPort>
  <MacAddress>00:50:56:9F:00:07</MacAddress>
  <OSGuess>Linux 2.6.X</OSGuess>
</Address>
```

Figure 17. Network entity XML after the Detect module.

## 2. Identify Module

The Identify module is where multiple features are combined to form a State. To accomplish this transformation, the exploit XML is opened up and all the features that create a state are pulled and placed into a vector. This matrix is then compared to the features of each IP address. Should a match occur, an entry is appended to the network entity XML of “State 0” and with an indicator of which vulnerability exists.

The same network entity after the Identify module is shown in Figure 18. The feature of port 80 being open and an OS of Linux have been fused to create a State 0 with vulnerability A.

```

<?xml version="1.0"?>
- <Address>
  192.168.120.3
  <TimeStamp>2013-06-21 09:28 Coordinated Universal Time</TimeStamp>
  <ProgramUsed>Nmap 6.25</ProgramUsed>
  <OpenPort>22</OpenPort>
  <OpenPort>80</OpenPort>
  <OpenPort>111</OpenPort>
  <OpenPort>5432</OpenPort>
  <OpenPort>6667</OpenPort>
  <OpenPort>9080</OpenPort>
  <OpenPort>9618</OpenPort>
  <MacAddress>00:50:56:9F:00:07</MacAddress>
  <OSGuess>Linux 2.6.X</OSGuess>
- <State>
  State 0
  <Vulnerability>A</Vulnerability>
  <TimeStamp>2013-06-21 09:31AM Pacific Daylight Time</TimeStamp>
</State>
</Address>

```

Figure 18. Network entity XML after the Identify module.

### 3. Predict Module

The Predict module is where the state is analyzed and the probabilities for which method of exploitation is recommended are determined. The first step the module takes is to retrieve all recommendations for each type of vulnerability and calculate their decimal percentage. For this proof-of-concept, probabilities are determined solely by previous successes divided by totals. The initial values were chosen to force the prediction towards a MiTM attack with Ettercap, but the operator feedback at the end of the React module eventually skews these numbers either higher or lower. Next, the module picks the attack with the highest probability and append this recommendation to the network entity XML.

Future work for expanding and making this section more complete is discussed in Chapter IV. The network entity file with the addition of recommendations for a MiTM attack using Ettercap is depicted in Figure 19. The information that this attack requires an additional vulnerability B to be present for an attack is also included.

```

<?xml version="1.0"?>
- <Address>
  192.168.120.3
  <TimeStamp>2013-06-21 09:28 Coordinated Universal Time</TimeStamp>
  <ProgramUsed>Nmap 6.25</ProgramUsed>
  <OpenPort>22</OpenPort>
  <OpenPort>80</OpenPort>
  <OpenPort>111</OpenPort>
  <OpenPort>5432</OpenPort>
  <OpenPort>6667</OpenPort>
  <OpenPort>9080</OpenPort>
  <OpenPort>9618</OpenPort>
  <MacAddress>00:50:56:9F:00:07</MacAddress>
  <OSGuess>Linux 2.6.X</OSGuess>
- <State>
  State 0
  <Vulnerability>A</Vulnerability>
  <TimeStamp>2013-06-21 09:31AM Pacific Daylight Time</TimeStamp>
</State>
- <Recommend>
  MiTM-Ettercap
  <Timestamp>2013-06-21 09:33AM</Timestamp>
  <Prob>0.625</Prob>
  <Requires>B</Requires>
</Recommend>
</Address>

```

Figure 19. Network entity XML after the Predict module.

#### 4. React Module

The React module is where the inputs from the previous modules are used to launch an actual attack against the target network or computer. This is done first by finding the recommendations of each network entity and determining which has the highest probability associated with their recommendation. The module then checks the recommendation to see if any further requirements exist. If additional requirements exist, then it searches the other network entities to find the required vulnerability. If none are found, then it moves to the network entity with the next highest probability of success associated with its recommendation. If the additional requirement is found, then it provides all of the necessary information to the function which launches the recommended attack. In this occurrence Ettercap was chosen and the command launched was:

```
ettercap.exe -T -M arp:remote /192.168.120.10/ /192.168.120.3/
```

This command launches a MiTM attack (-M) using ARP Poisoning between the network's webserver (192.168.120.10) and the target (192.168.120.3). After the

command is launched, the network entity XML is updated with the timestamp and attack launched. Feedback for success can then be input by the operator, which then updates the exploit XML accordingly. Determination of success or failure is further discussed in the next section.

Figure 20 an illustration of the final network entity XML created which depicts that the updated information for the attack performed and timestamp has been added to the network entity XML.

```

<?xml version="1.0"?>
- <Address>
  192.168.120.3
  <TimeStamp>2013-06-21 09:28 Coordinated Universal Time</TimeStamp>
  <ProgramUsed>Nmap 6.25</ProgramUsed>
  <OpenPort>22</OpenPort>
  <OpenPort>80</OpenPort>
  <OpenPort>111</OpenPort>
  <OpenPort>5432</OpenPort>
  <OpenPort>6667</OpenPort>
  <OpenPort>9080</OpenPort>
  <OpenPort>9618</OpenPort>
  <MacAddress>00:50:56:9F:00:07</MacAddress>
  <OSGuess>Linux 2.6.X</OSGuess>
- <State>
  State 0
  <Vulnerability>A</Vulnerability>
  <TimeStamp>2013-06-21 09:31AM Pacific Daylight Time</TimeStamp>
</State>
- <Recommend>
  MiTM-Ettercap
  <Timestamp>2013-06-21 09:33AM</Timestamp>
  <Prob>0.625</Prob>
  <Requires>B</Requires>
</Recommend>
- <AttackPerformed>
  MiTM-Ettercap
  <TimeStamp>2013-06-21 09:35AM</TimeStamp>
</AttackPerformed>
</Address>

```

Figure 20. Network entity XML after the React module.

## F. DETERMINING SUCCESS

The MiTM attack using ARP poisoning is accomplished in a few steps that are possible to observe using a monitoring program such as Wireshark. The first step is the

ARP Poisoning itself. This is observed by seeing ARP messages sent to the MAC addresses of the two target computers. Because it is within a single network, only the MAC addresses need to be manipulated to intercept the data. The IP addresses of both targets remain unchanged through the attack. As seen in Figure 21, the attack box (ending in 00:06) sends ARP messages to both of the victims (ending in 00:07 and 00:05) that the IP Address of the other victim belongs to the entity with the attack box MAC Address. This is considered ARP Poisoning of the network because now both victims have this information stored in their ARP cache.

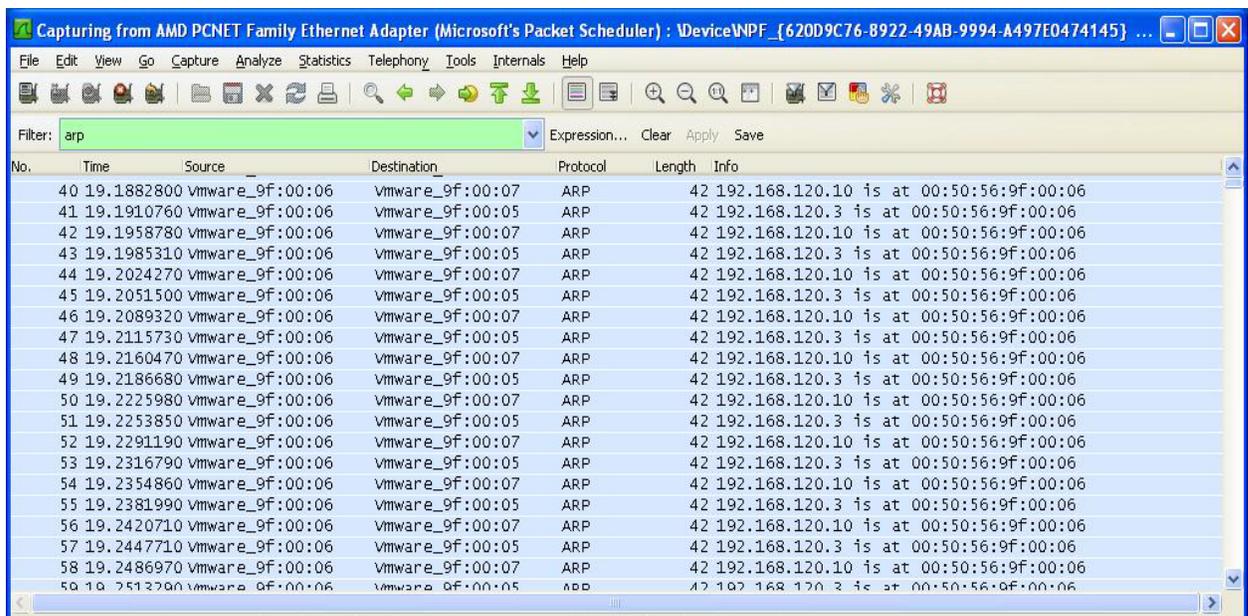


Figure 21. Wireshark capture of ARP poisoning the network via Ettercap.

A representation of the before and after source/destination IP and MAC addresses of the victims is shown in Figure 22. The attack box is now successfully intercepting any information passing between the two victims.

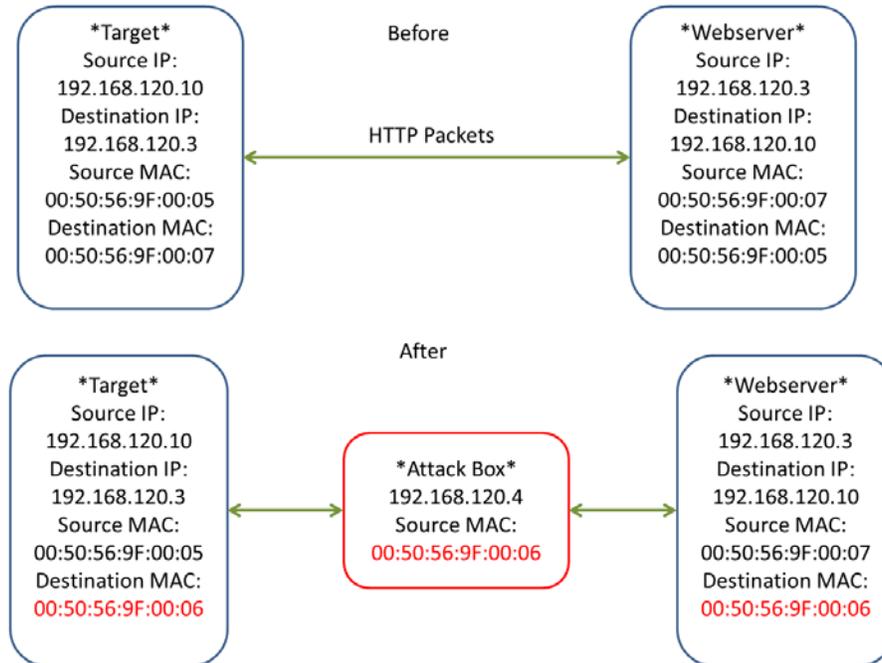


Figure 22. MiTM attack in the network.

Figure 23 is an illustration of the Wireshark output viewing the traffic between the two computers. This is an example of viewing a simple webpage being accessed by the target. If it was desired to watch similar traffic between the target and a webserver not on the network, the MiTM would be launched between the gateway router and the target.

No.	Time	Source	Destination	Protocol	Length	Info
22	19.1278240	192.168.120.10	192.168.120.3	ICMP	42	Echo (ping) request id=0x7ee7, seq=32487/59262, ttl=64
23	19.1307620	192.168.120.3	192.168.120.10	ICMP	42	Echo (ping) request id=0x7ee7, seq=32487/59262, ttl=64
11717	35.5856870	192.168.120.10	192.168.120.3	TCP	62	bridgecontrol > http [SYN] seq=0 win=16384 Len=0 MSS=1460
14527	38.5429420	192.168.120.10	192.168.120.3	TCP	62	bridgecontrol > http [SYN] seq=0 win=16384 Len=0 MSS=1460
15245	39.3025440	192.168.120.10	192.168.120.3	TCP	62	warmspotMgmt > http [SYN] seq=0 win=16384 Len=0 MSS=1460
17621	41.8122420	192.168.120.10	192.168.120.3	TCP	62	rdrmsmc > http [SYN] seq=0 win=16384 Len=0 MSS=1260 SA=192.168.120.10
20401	44.7773490	192.168.120.10	192.168.120.3	TCP	62	rdrmsmc > http [SYN] seq=0 win=16384 Len=0 MSS=1260 SA=192.168.120.10
25254	50.7928660	192.168.120.10	192.168.120.3	TCP	62	rdrmsmc > http [SYN] seq=0 win=16384 Len=0 MSS=1260 SA=192.168.120.10
35889	62.8361600	192.168.120.10	192.168.120.3	TCP	62	dab-sti-c > http [SYN] seq=0 win=16384 Len=0 MSS=1260 SA=192.168.120.10
38751	65.8866360	192.168.120.10	192.168.120.3	TCP	62	dab-sti-c > http [SYN] seq=0 win=16384 Len=0 MSS=1260 SA=192.168.120.10
44102	71.9022300	192.168.120.10	192.168.120.3	TCP	62	dab-sti-c > http [SYN] seq=0 win=16384 Len=0 MSS=1260 SA=192.168.120.10

Figure 23. Intercepted traffic viewed in Wireshark.

Should all of these indicators be observed, the operator would select “success” in the GUI; otherwise, “failure” would be selected. Further discussion on automating verification is discussed in Chapter IV.

## **G. CONCLUSIONS**

Final conclusions and results from the proof-of-concept are discussed in this section.

### **1. Intelligence Automation Success and Flexibility**

The proof-of-concept itself accomplished the goals and tasks laid out in Chapter II while only implementing a very limited version of the system. The groundwork was laid for scalability and expansion.

### **2. Stealth**

By using only the freeware programs to access the network there is no indication to an observer on the network that the process was automated.

### **3. Standardization and Scalability**

Standardization of freeware programs which are generated by a wide selection of individuals on the internet is impossible, but by using the Detect module as a translator program it is possible to integrate them. Ensuring the network entity XML fields and the exploit XML fields match is integral to this concept.

### **4. Summary**

The proof-of-concept implementation of the DIPR automated intelligence model for cyber penetration testing was explained in this chapter. The physical and virtual components as well as the required software were discussed as were the results of each module and how they interacted with the entire system. Future work, improvements and final conclusions are discussed in Chapter IV.

THIS PAGE INTENTIONALLY LEFT BLANK

## **IV. IMPROVEMENTS, FUTURE WORK AND CONCLUSIONS**

Potential improvements for this specific proof-of-concept, future work which would create a more robust product and final conclusions on the success of the overall system are covered in this chapter.

### **A. IMPROVEMENTS TO THE PROOF-OF-CONCEPT**

After the proof-of-concept was completed and tested, there were a few aspects that, if employed differently from the start, would have resulted in greater ease of implementation as well as a superior quality end product. While the current proof-of-concept meets the intended objectives, there are specific items that would make the process smoother.

#### **1. Operating Environment**

By coding in a Windows environment, the implementation of the freeware programs and meeting the program requirements of command line inputs and parseable outputs were difficult to meet. In general, Windows is more user-friendly, GUI focused, and in most cases, freeware programs lacked command line inputs or a parseable output. Nmap and Ettercap were two of the few programs that met these needs. By instead using a Linux environment for coding, the available number of suitable freeware programs would be much higher. A breakdown of several freeware programs that were examined during implementation is shown in Table 2. While these are by no means all of the available freeware programs, the breakdown was indicative of what is typically available. Many of the programs for Windows either lacked the command line input or the parseable output ability common in Linux based programs.

Table 2. Comparative look at requirements and operating systems in freeware programs.

	<b>Parseable Output</b>	<b>Command Line Input</b>	<b>Discovery</b>	<b>Attack</b>
<b>Windows</b>				
NMap	X	X	X	
Ettercap (XP)	X	X	X	X
Wireshark	X			
tshark	X	X		
Tetherreal (XP) *Command line version of Ethereal *	X	X	X	X
tcpdump	X			
NetStumbler (Wifi)	X	X	X	
Cain & Able				X
Nighthawk	X		X	X
<b>Linux</b>				
NMap	X	X	X	
Ettercap	X	X	X	X
Wireshark	X	X		
Xprobe2	X	X	X	
Cain & Able	X	X	X	X
tcpdump	X	X		
Ethereal	X	X	X	
dsniff	X	X	X	X
Kismet (Wifi)	X	X	X	
EtherApe	X	X	X	
Netcat	X	X	X	X

## 2. Robustness

Due to the limitations of the operating environment, only one of each type of program was implemented. To fully realize the usefulness of standardization, multiple types of each program should be included.

## B. FUTURE WORK

Future work to develop and expand upon the work successfully conducted in this thesis is discussed in this section. Recommendations for building in an additional assessment module, statistically modeling the predict module and improving the user friendliness of the GUI are discussed.

## 1. Additional Assessment Module

An additional module that could be added to the overall DIPR module is a standalone feedback module that would be able to tap into each of the four modules separately. This would allow a measure of effectiveness for each particular stage and could help troubleshooting and streamlining the entire process. This would allow for on-the-fly learning and troubleshooting. If any of the four modules are misconfigured or if the exploits themselves are incorrect, a standalone assessment module would be useful to find the issue and highlight where corrections need to be made. The implementation of this is shown by Figure 24.

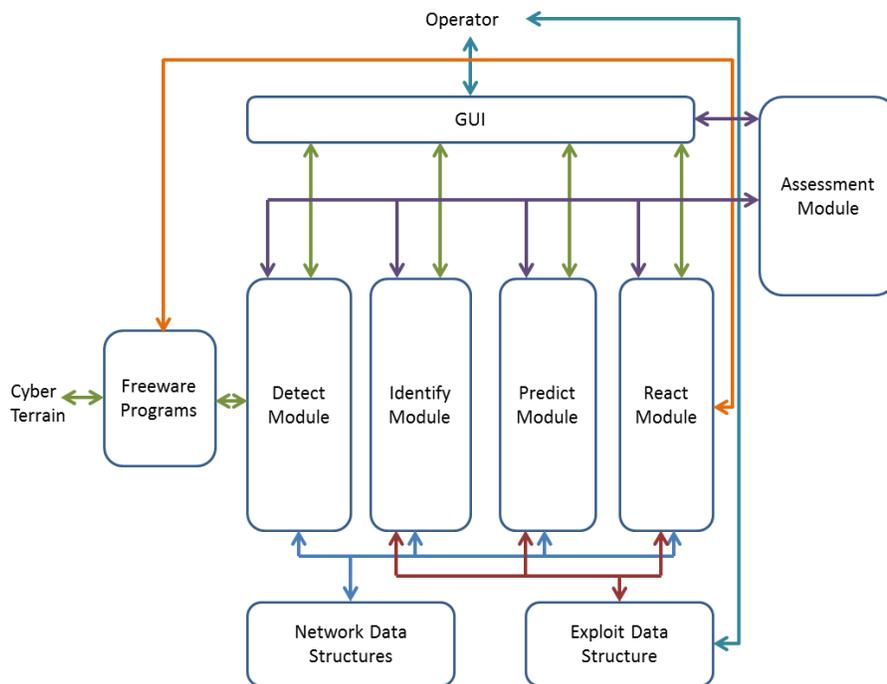


Figure 24. DIPR internal diagram with additional Assessment module.

By setting the Assessment module outside of operational flows but with full access to each module, the additional module will provide automated oversight and trouble shooting. Particularly because of the need for standardization of the XML files fields, simple typos or lack of coding expertise during the detect portion for a new freeware implementation could result in the system either not working or in a duplication of effort. Because of the nature of the automation, each module could cause the entire

system to break down, and the assessment module could help resolve exceptions or errors produced by the individual modules. This would reduce the time spent troubleshooting and provide a native metric of effectiveness for overall system health.

Additionally, failures and exceptions created by the attack program could be fed into the Assessment module discussed in the previous future works section to provide real time feedback and see if additional information needs to either be cultivated from the discovery program or if the parameters stated in the exploit document need to be altered.

## **2. Predict Module**

The Predict module provides the recommendation for type and implementation of cyber penetration attack launched by the React module. By developing and implementing a statistical model for which penetration attacks are the most likely to succeed based on network configuration, the Predict module can become the keystone of the system. As implemented (and as discussed in the previous subsection) in the proof-of-concept, the Predict module is simply a less-than or greater-than comparison between success ratios. This method is weak because only one implementation of cyber penetration attack will ever be generated, and it does not provide the opportunity for other implementations to improve their ratio. Nor does it provide any feedback for how or why those other implementations failed. Future work might include developing a model to see which implementations work the best and what network characteristics give them the greatest chance of success. This can then be used to populate the exploit XML with more detailed and precise network/host requirements.

## **3. GUI**

A robust GUI aids the operator in launching the desired attack against the network. Automation should be limited only to the implementation of what the operator desires; otherwise, the consequences of starting the system cannot be determined beforehand. Creating a GUI that informs the operator of options along the way can also be useful in the event that the system is placed onto the network with no prior knowledge of components. Striking the balance between providing flexibility and options while not overwhelming the operator with information is a delicate task that can be eased by a user-friendly advanced GUI.

#### **4. Intelligent Automation of New Freeware**

A valuable tool to reduce the amount of manual work for the operator would be automating the addition of new programs with a separate system. By being able to parse or determine the various options (either by trial and error or by options via the instructional page) and creating rules to determine which programs function together, new freeware can easily be imported into the system. However, this relies on accurate and robust documentation of all options for the freeware program, a factor that is unlikely to occur in all cases. Therefore, it does not eliminate the operator or the need for some manual coding but would save vast amounts of time and effort if implemented properly.

### **C. CONCLUSIONS**

Overall conclusions of both the DIPR system as a whole as well as the proof-of-concept specifically are presented in this section.

#### **1. DIPR System**

Overall, the Automated DIPR Penetration Testing System showed a unique and scalable method for automation of penetration testing by integrating freeware programs into a unified whole. By utilizing the DIPR intelligence automation model as a backbone architecture, integration was possible and showed a versatile adaptation which could be used to assimilate many other programs. Using individual XML documents for each entity on a network and an additional XML document to specify desired vulnerabilities, we were able to show that outputs of one program could be used to feed into the input of a disparate program. By processing and manipulating the outputs of the discovery programs, we can mesh them into a more advanced structure than is possible from any single program.

#### **2. Proof-of-concept**

The proof-of-concept successfully implemented a barebones implementation of the Automated Penetration Testing DIPR architecture. Using NMap and Ettercap as the freeware programs, we were able to show that the output of one program, processed and standardized, could be the input to another program. The processing and standardization

between these steps is where the artificial intelligence played a role. The processing was done by a series of modules set to detect individual features of network entities and to then process and mesh those features into individual states. These states could be interpreted as vulnerabilities and be acted upon.

## APPENDIX A

### GUI Code

```
#####  
#Module: GUI  
#Purpose: Step through each module, allow input from user  
#to update the exploit xml, display steps  
#Main FCN: WalkThroughSequence  
#####  
import Tkinter as tk  
from Tkinter import *  
import tkMessageBox  
import Detect  
import Identify  
import Predict  
import React  
  
#Sets up the GUI#  
top = Tk()  
top.title("DIPR Automated Cyber Penetration System")  
top.geometry("200x250")#sets size and geometry  
  
#####  
#Function: WalkThroughSequence  
#Purpose: Walks through each module, Builds the GUI  
#Input: None  
#Output: v - Success or failure of the entire walk through  
#####  
def WalkThroughSequence():  
    text = Text(top, width = 25, height = 9, wrap = WORD)  
    text.insert(INSERT, 'Starting... \n')  
    text.grid(row = 1, column = 0)  
    y = Detect.Discover("NMap")#runs detect module  
    if y == 'Success':  
        text.insert(END, 'Detect Successful \n')  
        z = Identify.IdStates()#runs identify module  
        if z == 'Success':  
            text.insert(END, 'Identify Successful \n')  
            w = Predict.Predict() #runs predict module  
            if w == 'Success':  
                text.insert(END, 'Predict Successful \n')  
                (v, vul, atk) = React.React() #runs React Module  
                if v == 'Success':
```

```

    text.insert(END, 'React Successful.\n Please view Wireshark \n')
    #adds in the buttons for success or failure
    #runs the success or failure function located in React
    C = Button(top, text = 'Success?', command = lambda:
React.UpdateSuccess(vul, atk))
    D = Button(top, text = 'Failure?', command = lambda: React.UpdateFailure(vul,
atk))
    C.grid(row = 2, column = 0)
    D.grid(row = 3, column = 0)
    return v

```

```

B = Button(top, text ="Start Sequence", command = WalkThroughSequence)
B.grid(row = 0,column = 0)#puts the button into our window

```

```

top.mainloop()#starts and maintains the window until complete

```

### Detect Code

```

#####
#Module: Detect
#Purpose: Detect network entities and
#create XML files with features of Entity
#Main FCN: Discover
#####

```

```

import os
import subprocess
import xml.etree.ElementTree as ET

```

```

#####
#Function: Discover
#Purpose: Discover network entities parse information
#store parsed information in an XML file
#Input: Program - Name of program desired to be used to discover the network entities
#Output: Success - States that the program has completed
#####

```

```

def Discover(Program):
    #Calls the NMap Program and pipes the result into a file called infosearch.txt
    #-O means look for possible Operating systems. To do this it will scan for open ports as
well
    #if Program == "Nmap":
        #os.system('"C:\\Program Files\\Nmap\\nmap.exe" -O 192.168.120.0/24 >
infosearch.txt')

```

```

info = [];
y = -1;
timestamp = ''
file = ''
#Index will eventually be the master index XML file that will keep track of who is in
the network
Index = ET.Element('Index')
#IP will eventually become the network entity XML
IP = ET.Element('Address')

#Begin to parse the infosearch file with the piped information from NMap
for line in open('infosearch.txt', 'r'): #Walk through each line of the file
    if 'Starting' in line: #pulling the timestamp for each scan
        k = line.split(' at ')
        timestamp = k[1] #timestamp at which NMap was run
    if 'scan report' in line: #We've run into a new IP address make a new port list
        if y > -1:
            tree = ET.ElementTree(IP)
            tree.write(file, "us-ascii", xml_declaration = None, default_namespace = None,
method = 'xml')
            IP = IP.clear()
            IP = ET.Element('Address')
            k = line.split(' ')
            p = k[4].strip() #calling the strip fcn pulls white space of either side of the string
            file = 'IP' + p + '.xml' #creates a filename based off of IP address
            IP.text = k[4].strip()
            ind = ET.SubElement(Index, 'IP') #creates our top node IP
            ind.text = k[4].strip()
            time = ET.SubElement(IP, 'TimeStamp') #Fills in details
            time.text = timestamp.strip()
            prog = ET.SubElement(IP, 'ProgramUsed') #What program is being used
            prog.text = 'Nmap 6.25'
            y = y+1
        if 'MAC Address:' in line: #Looks at what the MAC address is for that IP
            k = line.split(' ')
            ma = ET.SubElement(IP, 'MacAddress')
            ma.text = k[2].strip()
        if 'open' in line: #Looks at what ports are Open
            k = line.split('/')
            op = ET.SubElement(IP, 'OpenPort')
            op.text = k[0].strip()
        if 'Running' in line: #Looks at what OS NMap thinks its running
            k = line.split(':')
            k = k[1].split(',')
            for item in k:

```

```

    if 'or ' in item and item.index('or ') < 4:
        h = item.split('or ')
        OSGuess = ET.SubElement(IP, 'OSGuess')
        OSGuess.text = h[1].strip()
    else:
        OSGuess = ET.SubElement(IP, 'OSGuess')
        OSGuess.text = item.strip()

#Prints our master index list and the XML list to their respective files
tree = ET.ElementTree(IP)
tree.write(filename, "us-ascii", xml_declaration = None, default_namespace = None,
method = 'xml')
tree = ET.ElementTree(Index)
tree.write('IPAddressIndex.xml', encoding = 'us-ascii')
return 'Success'

```

Discover("NMap")

#### Identify Code

```

#####
#Module: Identify
#Purpose: Go through each Network Entity and
#determine if a state exists
#Main FCN: IdStates()
#####

```

```

import xml.etree.ElementTree as ET
from datetime import datetime

```

```

#####
#Function: IsEqual
#Purpose: Compares an exploit list against a single IP
#address to determine if a state exists
#Input: a - exploit vector that looks like:
#[ 'A', ['OSGuess', 'Linux 2.6.X'], ['OpenPort', '80']]
#[ "Name of Vulnerability", [Vector of Aspect followed by specifics], etc..]
# b - pointer to the XML of a single IP address
#Output: matching- vector containing the name of vulnerability
# and if each thing checked were true or false. Will look like:
#[ 'A', False, False]
#####
def IsEqual(a , b):
    matching = []
    matching.append(a[0])

```

```

#check for matching ports
ports = b.findall('OpenPort')
check = []
for p in ports:
    u = p.text
    check.append(u)
counter = 0
ans = False
for y in a:
    if counter > 0:
        if y[0] == 'OpenPort':
            for i in y:
                for u in check:
                    if i == u:
                        ans = True
            counter = counter + 1
matching.append(ans)

```

```

#check for matching OSs
osg = b.findall('OSGuess')
check = []
for os in osg:
    u = os.text
    check.append(u)
counter = 0
ans = False
for y in a:
    if counter > 0:
        if y[0] == 'OSGuess':
            for i in y:
                for u in check:
                    if i in u:
                        ans = True
            counter = counter + 1
matching.append(ans)
return matching

```

#####

#Function: GetExploits

#Purpose: pulls the entire exploit xml and makes

#it a vector we can use to compare to the IP address

#Input: None

#Output: exploitlist vector contains a list of exploits and vectors for  
#for the various aspects. Looks like:

#[['A', ['OSGuess', 'Linux 2.6.X'], ['OpenPort', '80']],

```

# ['B', ['OSGuess', 'Microsoft Windows 2000|XP', 'Windows XP SP2'], ['OpenPort',
'3389', '6667', '7000']]
#####
def GetExploits():
    tree = ET.parse('Exploits.xml')
    root = tree.getroot()
    exploitlist = []
    l=0
    for Vul in root.iter('Vulnerability'):
        y = []
        exploitlist.append([Vul.text])
        for Asp in Vul.iter('Aspect'):
            y.append(Asp.text)
            for Sp in Asp.iter('Spec'):
                y.append(Sp.text)
            exploitlist[l].append(y)
            y =[]
        l = l+1;
    return exploitlist

#####
#Function: GetIPAddresses
#Purpose: pulls all IP addresses from the master list and returns a vector with them
#Input: None
#Output: vector list of IP addresses
#Looks like: ['192.168.120.3', '192.168.120.4', '192.168.120.10', '192.168.120.150']
#####
def GetIPAddresses():
    tree = ET.parse('IPAddressIndex.xml')
    root = tree.getroot()
    IPAddList = []
    for IP in root.iter('IP'):
        IPAddList.append(IP.text)
    return IPAddList

#####
#Function: IdStates
#Purpose: uses other functions to create queries, test if they are true
#and then updates the network entity xml to reflect that a state exists or not
#Input: None
#Output: "Success" to let the GUI know that Identify completed correctly
#####
def IdStates():
    elist = GetExploits()
    ilist = GetIPAddresses()

```

```

for add in ilet:
    file = 'IP' + add + '.xml'
    tree = ET.parse(file)
    root = tree.getroot()
    t = datetime.now().strftime("%Y-%m-%d %I:%M%p")
    for ex in elist:
        ans = IsEqual(ex, root)
        if all(ans):
            state = ET.SubElement(root, 'State')
            state.text = 'State 0'
            Vul = ET.SubElement(state, 'Vulnerability')
            Vul.text = ans[0]
            Time = ET.SubElement(state, 'TimeStamp')
            Time.text = t + " Pacific Daylight Time"
    tree.write(file, 'us-ascii')
return 'Success'

```

IdStates()

Predict Code

```

#####
#Module: Predict
#Purpose: Go through each Network Entity, find if a state
#exists, make a recommendation for an attack for that state
#provide expected probability of success
#Main FCN: Predict()
#####

import xml.etree.ElementTree as ET
from datetime import datetime

#####
#Function: GetRecommendations()
#Purpose: Opens exploits.xml, calculates recommended exploits
#and percentages
#Input: None
#Output: list of exploits and probabilities. Looks like:
#[['A', ['MiTM-Ettercap', 0.625, 'B'], ['MoTS-Cain and Able', 0.428]]]
#####

def GetRecommendations():
    tree = ET.parse('Exploits.xml')
    root = tree.getroot()
    exploitlist = []
    l=0

```

```

for Vul in root.iter('Vulnerability'):
    y = []
    exploitlist.append([Vul.text])
    for Rec in Vul.iter('Recommend'):
        y.append(Rec.text)
        for Pr in Rec.iter('Prob'):
            v = Pr.text
            v = v.split(':')
            n = float(v[0])/float(v[1])
            y.append(n)
        for Rq in Rec.iter('Requires'):
            y.append(Rq.text)
        exploitlist[l].append(y)
    y = []
    l = l+1;
return exploitlist

```

```

#####

```

```

#Function: GetIPAddresses
#Purpose: pulls all IP addresses from the master list and returns a vector with them
#Input: None
#Output: vector list of IP addresses
#Looks like: ['192.168.120.3', '192.168.120.4', '192.168.120.10', '192.168.120.150']
#####

```

```

def GetIPAddresses():
    tree = ET.parse('IPAddressIndex.xml')
    root = tree.getroot()
    IPAddList = []
    for IP in root.iter('IP'):
        IPAddList.append(IP.text)
    return IPAddList

```

```

#####

```

```

#Function: MakeRec
#Purpose: given a vector of found vulnerabilities choose the
#one with the highest prob and return it
#Input: r. List of exploits.
#Looks like: [['A', ['MiTM-Ettercap', 0.625, 'B']], ['MoTS-Cain and Able', 0.428]]
#General form is Name of Vulnerability, [Name of attack, probability, additional
requirements], etc
#Output: Exploit with recommended attack and probability and any other requirements
#Looks like: ['MiTM-Ettercap', 0.625, 'B']
#Note: most future work with Predict would be in here
#####

```

```

def MakeRec(r):

```

```

y = [' ', 0]
counter = 0
for a in r:
    if counter > 0:
        if a[1] > y[1]:
            y = a
        counter = counter + 1
return y

#####
#Function: Predict
#Purpose: Go through network entities, see if a state 0 exists, if it does find
#the recommended exploit, update the entity xml to reflect this
#Input: None
#Output: Success to indicate the module has run correctly
#####
def Predict():
    rlist = GetRecommendations()
    ilit = GetIPAddresses()
    t = datetime.now().strftime("%Y-%m-%d %I:%M%p")
    for add in ilit:
        filen = 'IP' + add + '.xml'
        tree = ET.parse(filen)
        root = tree.getroot()
        for st in root.iter("State"):
            if st.text == "State 0":
                for vul in st.iter('Vulnerability'):
                    for r in rlist:
                        if r[0] == vul.text:
                            z = MakeRec(r)
                            RElem = ET.SubElement(root, 'Recommend')
                            RElem.text = z[0]
                            TElem = ET.SubElement(RElem, 'Timestamp')
                            TElem.text = t
                            PElem = ET.SubElement(RElem, 'Prob')
                            PElem.text = str(z[1])
                            if len(z) > 2:
                                QElem = ET.SubElement(RElem, 'Requires')
                                QElem.text = z[2]
        tree.write(filen, 'us-ascii')
    return 'Success'
Predict()

```

## React Code

```
#####  
#Module: React  
#Purpose: Take recommendations from the network entities  
#verify required conditions exist. Launch attack. Update  
#success or failutre based on user input  
#Main FCN: React()  
#####  
import xml.etree.ElementTree as ET  
import subprocess  
from datetime import datetime  
  
#####  
#Function: GetVulnerabilities  
#Purpose: Finds vulnerabilities in each XML file and  
#returns them to the calling function along with their  
#probability and any requirements  
#Input: None  
#Output: (vlist, prob) vlist is a matrix containing  
#vectors of IP addresses with recommended attacks, the attack  
#and further requirements for the attack  
#vlist looks like: [['A', '192.168.120.3', 'MiTM-Ettercap', 'B'],  
#[ '192.168.120.10', 'DOS-Cain and Able'],  
#prob is a vector of probabilities of each of the vlist addresses  
#prob looks like: ['0.625', '0.6', '0.625']  
#####  
def GetVulnerabilities():  
    ilit = GetIPAddresses()  
    vlist = []  
    prob = []  
    for i in ilit:  
        filen = 'IP'+ i + '.xml'  
        tree = ET.parse(filen)  
        root = tree.getroot()  
        hold = []  
        for v in root.iter("State"):  
            for s in v.iter("Vulnerability"):  
                hold.append(s.text)  
        for rc in root.iter("Recommend"):  
  
            hold.append(root.text)  
            hold.append(rc.text)  
            for pr in rc.iter("Prob"):  
                prob.append(pr.text)
```

```

        for rq in rc.iter("Requires"):
            hold.append(rq.text)
            vlist.append(hold)
        return (vlist, prob)

#####
#Function: FindVul
#Purpose: Finds a specific vulnerability from all of the IP
#addresses. Used to see if additional requirements are met
#Input: fv - name of the vulnerability to be found i.e., 'B'
#Output: IP- returns first IP address of an entity with that
#vulnerability
#####
def FindVul(fv):
    ilit = GetIPAddresses()
    IP = 'No IP Found'
    for i in ilit:
        filen = 'IP' + i + '.xml'
        tree = ET.parse(filen)
        root = tree.getroot()
        for vul in root.iter('Vulnerability'):
            if vul.text == fv:
                IP = root.text
    return IP

#####
#Function: runEttercap
#Purpose: Runs the ettercap program from the command line
#Input: victim1 and victim2- IP addresses of the two entities we'd like to
#perform a MiTM attack on
#Output: True - denotes attack called
#####
def runEttercap(victim1, victim2):
    #command = ["C:\\Program Files\\Ettercap Development Team\\Ettercap-
0.7.4\\ettercap.exe", '-T', '-M', 'arp:remote', '/' + victim1 + '/', '/' + victim2 + '/']
    #subprocess.call(command)
    update(victim1, True, 'MiTM-Ettercap')
    return True

#####
#Function: update
#Purpose: Updates the entity xml with the time date and success
#(or failure) of the attack
#Input: IP - network entity IP address, Success- success or failure
#Attack - what attack was performed

```

```

#Output: True - denotes update complete
#####
def update(IP, Success, Attack):
    file = 'IP' + IP + '.xml'
    tree = ET.parse(file)
    root = tree.getroot()
    t = datetime.now().strftime("%Y-%m-%d %I:%M%p")
    if Success == True:
        AElem = ET.SubElement(root, 'AttackPerformed')
        AElem.text = Attack
        TElem = ET.SubElement(AElem, 'TimeStamp')
        TElem.text = t
    tree.write(file, 'us-ascii')
    return True

#####
#Function: GetIPAddresses
#Purpose: pulls all IP addresses from the master list and returns a vector with them
#Input: None
#Output: vector list of IP addresses
#Looks like: ['192.168.120.3', '192.168.120.4', '192.168.120.10', '192.168.120.150']
#####
def GetIPAddresses():
    tree = ET.parse('IPAddressIndex.xml')
    root = tree.getroot()
    IPAddList = []
    for IP in root.iter('IP'):
        IPAddList.append(IP.text)
    return IPAddList

#####
#Function: React
#Purpose: gets list of possible attacks, probabilities and requirements
#finds highest probability, if requirements are met it launches that attack
#if they are not it goes to the next highest probability and continues to check
#until it can launch. Then it updates.
#Input: None
#Output: Success- tells GUI that react module completed
#####
def React():
    (vlist, prob) = GetVulnerabilities()
    p = prob.index(max(prob))
    attack = vlist[p]
    if len(attack) > 3:
        v2 = FindVul(attack[3])

```

```

if attack[2] == 'MiTM-Ettercap':
    runEttercap(attack[1], v2)
if attack[2] == 'DOS-Cain and Able':
    print 'No current attack...sorry'
return ('Success', attack[0], attack[2])

#####
#Function: UpdateSuccess
#Purpose: Updates Exploit XML probability for that attack with
#success.
#Input: Vul - vulnerability exploited
#Attack - attack launched for that vulnerability
#Output: 'Updated' to indicate that update has completed
#####
def UpdateSuccess(Vul, Attack):
    tree = ET.parse("Exploits.xml")
    root = tree.getroot()
    for V in root.iter("Vulnerability"):
        if V.text == Vul:
            for A in V.iter("Recommend"):
                if A.text == Attack:
                    for P in A.iter("Prob"):
                        k = P.text.split(":")
                        k[0] = int(k[0])+1
                        k[1] = int(k[1])+1
                        m = str(k[0])+ ":" + str(k[1])
                        P.text = m
    tree.write("Exploits.xml", 'us-ascii')
    return 'Updated'

#####
#Function: UpdateFailure
#Purpose: Updates Exploit XML probability for that attack with
#failure.
#Input: Vul - vulnerability exploited
#Attack - attack launched for that vulnerability
#Output: 'Updated' to indicate that update has completed
#####
def UpdateFailure(Vul, Attack):
    tree = ET.parse("Exploits.xml")
    root = tree.getroot()
    for V in root.iter("Vulnerability"):
        if V.text == Vul:
            for A in V.iter("Recommend"):
                if A.text == Attack:

```

```
for P in A.iter("Prob"):
    k = P.text.split(":")
    k[1] = int(k[1])+1
    m = k[0]+ ":" + str(k[1])
    P.text = m
tree.write("Exploits.xml", 'us-ascii')
return 'Updated'
```

React()

## APPENDIX B

Exploit.xml

```
<Exploits>
  <Vulnerability>A
    <Aspect>OSGuess
      <Spec>Linux 2.6.X</Spec>
    </Aspect>
    <Aspect>OpenPort
      <Spec>80</Spec>
    </Aspect>
    <Recommend>MiTM-Ettercap
      <Prob>5:8</Prob>
      <Requires>B</Requires>
    </Recommend>
    <Recommend>MoTS-Cain and Able
      <Prob>3:7</Prob>
    </Recommend>
  </Vulnerability>
  <Vulnerability>B
    <Aspect>OSGuess
      <Spec>Microsoft Windows 2000|XP</Spec>
      <Spec>Windows XP SP2</Spec>
    </Aspect>
    <Aspect>OpenPort
      <Spec>3389</Spec>
      <Spec>6667</Spec>
      <Spec>7000</Spec>
    </Aspect>
    <Recommend>MiTM-Ettercap
      <Prob>5:10</Prob>
    </Recommend>
    <Recommend>DOS-Cain and Able
      <Prob>6:10</Prob>
    </Recommend>
  </Vulnerability>
</Exploits>
```

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF REFERENCES

- [1] E. Chow, "Ethical hacking & penetration testing," University of Waterloo, Waterloo, Canada, No. AC 626, 2011.
- [2] B. Jurjonas, "Smart selection and configuration of cyber sensors for active defensive cyber operations," M.S. thesis, Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA, March 2012.
- [3] "Network operations," class notes for CY 3602, Department of Electrical and Computer Engineering, Naval Postgraduate School, winter 2012.
- [4] D. Goshorn, "Cyber security and cyber warfare in a network-centric smart sensor system of systems," brief given to RADM William Leigher. Naval Postgraduate School, Monterey, CA, Spring 2010.
- [5] D. Goshorn, and R. Goshorn, "Cyber security in a network-centric smart-environment system of systems" presented at the Cyber Summit Conference. Monterey, CA, 2009.
- [6] Assistant Secretary of Defense for Networks and Information Integration/Department of Defense Chief Information Officer, "Information assurance workforce improvement program (implementing change 3, January 24, 2012)," Department of Defense, Washington DC, No. DoD 8570.01-M, 2005.
- [7] J. Donaldson. (2012, April 25). "U.S. Fleet Cyber Command/U.S. Tenth Fleet/Home," FCC/C10F N62. [Online]. Available: <http://www.fcc.navy.mil/>.
- [8] Dawnbreaker Inc., (2010, Nov. 3). "Phase III Portal". [Online]. Available: <http://www.dawnbreaker.com/portals/p3p/opnav/opnav-n2-n6.php>
- [9] PEO C4I PMW 130, "SPAWAR," PEO C4I, 7 July 2010. [Online]. Available: [http://www.public.navy.mil/spawar/Press/Pages/07212010\\_PMW130.aspx](http://www.public.navy.mil/spawar/Press/Pages/07212010_PMW130.aspx)
- [10] J. P. McDermott, "Attack net penetration testing," in *Proc. 2000 workshop on new security paradigms*, New York, pp. 15–21.

- [11] R. Goshorn, D. Goshorn, J. Goshorn, and L. Goshorn. "The need for distributed intelligence automation implemented through four overlapping approaches. Intelligence Automation Software, Standardization for Interoperability, Network-Centric System of Systems Infrastructure (with Advanced Cloud Computing) and Advanced Sensors" Center of Excellence in Command, Control, Communications, Computing and Intelligence. 2011. [Online]. Available: <http://c4i.gmu.edu/events/reviews/2011/papers/12-Goshorn-paper.pdf>
- [12] A. Bodhani, "Bad in a good way [information technology security]" *Engineering & Technology*, vol.7, no.12, pp.64, 68, Jan. 2013.
- [13] D. Rohret and A. Jett, "Red teaming: hacking techniques for IT professionals," Aardvark, Salt Lake City, UT, 2005.
- [14] A. Vakali, B. Catania, and A. Maddalena, "XML data stores: Emerging practices," *IEEE Internet Computing*, pp. 62–69, 2005.
- [15] N. Samant, "Automated penetration testing," M.S. project (Paper 180), San Jose State University, San Jose, CA, 2011.
- [16] O. H. Kwon *et al*, "HackSim: An automation of penetration testing for remote buffer overflow vulnerabilities," in *Information Networking: Convergence in Broadband and Mobile Networking*, C. Kim, Ed., Hiedleberg, Springer-Verlag Berlin, pp. 652-661, 2005.
- [17] B. Duan, Y. Zhang and D. Gu, "An easy-to-deploy penetration testing platform," in *The 9th International Conference for ICYCS*, Hunan, China, 2008.
- [18] G. Lyon. (n.d.). "Intro," [Online]. Available: [www.nmap.org](http://www.nmap.org)
- [19] A. Ornaghi and M. Valleri. (2013). The Ettercap Project. [Online]. Available: <http://ettercap.github.io/ettercap/about.html>
- [20] Rapid 7. (2013). Metasploit about. [Online]. Available: <http://www.metasploit.com/about/>
- [21] M. Montoro. (n.d.). Projects [Online]. Available: <http://www.oxid.it/cain.html>
- [22] D. E. F. Gehringer. (n.d.) Ethics in computing:abuse: Denial of service attack. [Online]. Available: <http://ethics.csc.ncsu.edu>

## INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center  
Ft. Belvoir, Virginia
2. Dudley Knox Library  
Naval Postgraduate School  
Monterey, California