



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**ATTACK GRAPHS FOR MODELING AND
SIMULATING SOPHISTICATED CYBER ATTACKS**

by

Travis L. Swiatocha

June 2018

Thesis Advisor:

Alan B. Shaffer

Co-Advisor:

Gurminder Singh

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<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.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 2018	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE ATTACK GRAPHS FOR MODELING AND SIMULATING SOPHISTICATED CYBER ATTACKS			5. FUNDING NUMBERS	
6. AUTHOR(S) Travis L. Swiatocha				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES 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.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) The growing importance of the cyber domain to the military has created a need not only to train its cyber operators, but also to provide an environment for them to plan, develop, and rehearse cyber-attacks to determine their effectiveness. The Malicious Activity Simulation Tool (MAST) is a Naval Postgraduate School developed application designed to simulate cyber-attack scenarios on adversary networks. This thesis extends the capabilities of MAST by enabling the development of sophisticated cyber-attack scenarios. We define a methodology for formally modeling cyber-attacks, simulating their execution, and observing their effects on virtualized adversary networks. Our methodology decomposes a cyber-attack graph into atomic events, represented as a finite state machine. We simulate the execution of the state machine utilizing MAST on a virtualized adversary network, which allows us to observe the entire attack sequence, and the effects achieved on the target by the attack. We demonstrate our methodology stepping through the attack development from its high level objectives, down to its state machine that we simulate utilizing MAST. Finally, we demonstrate our ability to successfully simulate a sophisticated denial-of-service attack scenario on an adversary target.				
14. SUBJECT TERMS cyber-attack, MAST, attack graph, attack model, simulation, modeling, offensive cyber operations			15. NUMBER OF PAGES 99	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**ATTACK GRAPHS FOR MODELING AND SIMULATING SOPHISTICATED
CYBER ATTACKS**

Travis L. Swiatocha
Captain, United States Marine Corps
BS, U.S. Naval Academy, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
June 2018**

Approved by: Alan B. Shaffer
Advisor

Gurminder Singh
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The growing importance of the cyber domain to the military has created a need not only to train its cyber operators, but also to provide an environment for them to plan, develop, and rehearse cyber-attacks to determine their effectiveness. The Malicious Activity Simulation Tool (MAST) is a Naval Postgraduate School–developed application designed to simulate cyber-attack scenarios on adversary networks. This thesis extends the capabilities of MAST by enabling the development of sophisticated cyber-attack scenarios. We define a methodology for formally modeling cyber-attacks, simulating their execution, and observing their effects on virtualized adversary networks.

Our methodology decomposes a cyber-attack graph into atomic events, represented as a finite state machine. We simulate the execution of the state machine utilizing MAST on a virtualized adversary network, which allows us to observe the entire attack sequence, and the effects achieved on the target by the attack. We demonstrate our methodology stepping through the attack development from its high level objectives, down to its state machine that we simulate utilizing MAST. Finally, we demonstrate our ability to successfully simulate a sophisticated denial-of-service attack scenario on an adversary target.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	PROBLEM STATEMENT	2
	1. Primary Question.....	2
	2. Secondary Questions.....	2
B.	SCOPE	2
C.	BENEFITS OF STUDY	3
D.	THESIS OUTLINE.....	3
	1. Chapter II: Background.....	3
	2. Chapter III: Condition-Based Execution and Simulation	3
	3. Chapter IV: Implementation and Testing.....	4
	4. Chapter V: Conclusion and Future Work.....	4
II.	BACKGROUND	5
A.	OFFENSIVE CYBERSPACE OPERATIONS	5
B.	JOINT TARGETING CYCLE	5
	1. Phase 1: The End State and Commander’s Objectives.....	6
	2. Phase 2: Target Development and Prioritization.....	6
	3. Phase 3: Capabilities Analysis	6
	4. Phase 4: Commanders Decision and Force Assignment	7
	5. Phase 5: Mission Planning and Force Execution	7
	6. Phase 6: Target Assessment	7
C.	CYBER RECONNAISSANCE	8
	1. Passive Collection Methods.....	9
	2. Active Collection Methods	10
	3. Social Engineering	11
D.	TARGET SYSTEM AND VULNERABILITY ANALYSIS	12
	1. Network Topology.....	12
	2. Enumeration.....	13
	3. Host Physical Environment and Purpose	14
	4. Attack Graph.....	14
E.	MALICIOUS ACTIVITY SIMULATION TOOL	16
	1. MAST Architecture	16
	2. Example MAST Scenario	17
	3. Simulating Cyberspace Operations.....	19
F.	CHAPTER SUMMARY.....	19

III.	CONDITION-BASED MALWARE EXECUTION AND SIMULATION	21
A.	INITIAL CONDITIONS.....	23
1.	System Access	23
2.	Vulnerability.....	24
3.	Payload.....	25
B.	CONDITIONS.....	27
1.	System Conditions.....	30
2.	User Conditions	31
3.	Attacker-Defined Conditions	31
C.	CYBER EFFECTS AND MAST SIMWARE	33
D.	CURRENT MAST DESIGN AND METHODOLOGY	34
E.	CHAPTER SUMMARY.....	35
IV.	IMPLEMENTATION AND TESTING.....	37
A.	IMPLEMENTATION	37
1.	Scenario.....	37
2.	Attack Graph.....	39
3.	Attack Model	43
4.	Module Development	45
B.	TESTING.....	50
1.	Cyber Battle Lab.....	50
2.	Target Network	51
3.	Attack Scenario Scripting	52
4.	Attack Simulation	57
C.	SUMMARY	61
V.	CONCLUSIONS AND FUTURE WORK.....	63
A.	SUMMARY	63
B.	CONCLUSIONS	63
C.	RECOMMENDATIONS FOR FUTURE WORK.....	64
1.	Graphical User Interface for Scenario Scripting	64
2.	Client Program Improvement	65
3.	Additional Modules.....	65
4.	MAST GUI	65
	APPENDIX A. TARGET SPECIFIC OS SOURCE CODE.....	67
	APPENDIX B. TARGET USER SOURCE CODE	69

APPENDIX C. APP RUNNING SOURCE CODE	71
APPENDIX D. INET CONNECTION SOURCE CODE	73
APPENDIX E. LOCK OUT SOURCE CODE.....	75
LIST OF REFERENCES	77
INITIAL DISTRIBUTION LIST	79

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1.	Example Network Topology	13
Figure 2.	Attack Graph for a Test Network. Source: Jajodia, Sushil and Noel (2010).	15
Figure 3.	Three-Tiered MAST Architecture. Source: Aybar (2017).	17
Figure 4.	Example MAST Implementation	18
Figure 5.	Finite State Machine	22
Figure 6.	Initial Condition State Machine	23
Figure 7.	Network Topology for Alice’s Attack	28
Figure 8.	Attack Model for Alice’s Attack Path	29
Figure 9.	Stuxnet Export 15 Control Flow. Source: Falliere, Murchu, and Chen (2011).	32
Figure 10.	Target Network Topology	38
Figure 11.	Target Network Attack Graph	42
Figure 12.	Attack Model for the Scenario Attack Path	44
Figure 13.	LockOut SimWare Module	49
Figure 14.	CYBL Test Network	52
Figure 15.	Test Scenario Script	56
Figure 16.	MAST GUI Prior to the Scenario Start	57
Figure 17.	Phishing Email Sent to Target Network Users	58
Figure 18.	Attacker’s Web Server and Download Warning	59
Figure 19.	MAST GUI with All Targets Infected	60
Figure 20.	Complete Event Log with a Single Host Infected	60

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Transition Table for Alice’s Attack Path	30
Table 2.	Operating Systems, Ports and Services.....	39

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ACL	Access Control List
BDA	Battle Damage Assessment
CDA	Collateral Damage Assessment
CIA	Confidentiality, Integrity, Availability
CND	Computer Network Defense
CVE	Common Vulnerability and Exposure
CYBL	Cyber Battle Lab
DCO	Defensive Cyberspace Operations
DNS	Domain Name Service
DOD	Department of Defense
DODIN	Department of Defense Information Network Operations
EICAR	European Institute for Computer Anti-Virus Research
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IMAP	Internet Message Access Protocol
IOT	Internet of Things
IT	Information Technology
ISR	Intelligence, Surveillance, Reconnaissance
JDK	Java Development Kit
JRE	Java Runtime Environment
JTC	Joint Targeting Cycle
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MAST	Malicious Activity Simulation Tool
MSF	Metasploit Framework
NAT	Network Address Translation
NFC	Near Field Communication
NIDS	Network Intrusion Detection System
NIST	National Institute of Standards and Technology

OCO	Offensive Cyberspace Operations
OS	Operation System
SES	Scenario Execution Server
SGS	Scenario Generation System
SMTP	Simple Mail Transfer Protocol
TCP	Transport Control Protocol
WLAN	Wireless Local Area Network

ACKNOWLEDGMENTS

I would like to thank my thesis advisors, Dr. Alan Shaffer, and Dr. Gurminder Singh, for their guidance and commitment over the last two years. Thank you for your time and leadership, and for sharing your immense knowledge and experience within this field.

I would also like to thank all those who have played a role in the development of MAST. A special thanks goes to Lou Aybar for getting me interested in MAST. Your research and (well-documented) hard work laid the foundation that I was able to build on.

Most importantly, I would like to thank my wife, Caroline. Thank you for your patience, support and love, and for taking care of everything while I was locked away coding and writing. I could not have done this without you. I love you.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The magnitude to which the Internet has enabled and expanded the capabilities of the Department of Defense (DoD) cannot be overstated. From information sharing to command and control of systems in a global operating environment, the Internet has extended lines of communications and expanded the ability to project power through an entirely new warfare domain. The rapid growth and evolution of cyberspace, from the growing trend toward smart technologies and the Internet of Things (IoT), to supporting the nation's critical infrastructure, have made protecting the cyber domain from attacks a necessity for DoD.

As in the physical warfare domains of land, air, maritime, and space, simply defending the cyberspace domain is not sufficient. Maintaining the capability to conduct offensive cyberspace operations is just as critical as defensive cyberspace operations for deterrence or employing cyber weapons when necessary. Cyber weapons must have the capability to “degrade, disrupt, or destroy access to, operation of, or availability of a target by a specified level for a specified time period” (Joint Chiefs of Staff 2013a, 26). Cyber weapons that are employed as part of a larger integrated fires plan require that the weapons' effects must be achieved before a phase can be completed or a unit can maneuver. In order to accomplish this, offensive cyber teams must be well trained, and cyberspace attacks must be meticulously planned and coordinated.

Training in the physical domains is built on years of battle tested doctrine and tactics, and conducted on ranges that replicate the enemy, the operating environment and enemy tactics, weapons and surroundings. This enables planning and development of courses of action, and identification of risks and shortfalls. It can also identify enemy vulnerabilities which can be exploited, or areas that must be avoided to prevent detection.

The concept of training and planning in the cyberspace domain should be no different. Cyberspace attacks must be planned in a realistic environment that replicates an adversary network as much as intelligence will permit. The interconnected nature of the cyberspace domain means that an untested attack may have unintended effects that result

in collateral damage to third-party networks, or worse, to friendly networks. This amplifies the necessity of a tool like MAST, where Aybar (2017), Belli (2016) and Lowney (2015) have demonstrated its ability to bridge the gap in training and planning in the cyberspace domain. Recent highly publicized successful attacks and security breaches, such as WannaCry and Mirai, reinforce the necessity of establishing dominance in cyberspace and the importance of effective and realistic training of cyberspace warriors to achieving that end state.

A. PROBLEM STATEMENT

This thesis addresses the following research questions.

1. Primary Question

How can offensive cyberspace attacks be modeled to effectively simulate and rehearse their execution against a target network?

2. Secondary Questions

What techniques can be used to simulate the effects of a denial-of-service attack on an adversary network?

What techniques can be used to simulate the sequencing of offensive cyberspace operations?

B. SCOPE

This thesis leverages and extends previous research conducted on MAST. Past research has demonstrated the ability to successfully simulate attacks on a simple adversary network, the ability to target a specific host identified by MAC address or IP address, and the ability to perform temporal sequencing of an attack through a Logic Bomb or Attack while Idle scenario (Aybar 2017).

The focus of this thesis is the development of a model to accurately represent the execution of sophisticated cyberspace attacks so they can be simulated and rehearsed against a virtualized adversary network. This thesis demonstrates the ability of MAST to

simulate a cyberspace attack scenario based on the model, and develops additional SimWare and modules that add to the MAST inventory. Additionally, this research includes the development of a more realistic target network for future MAST development and testing.

C. BENEFITS OF STUDY

MAST has demonstrated its capability as a platform for simulating cyberspace attacks for both offensive and defensive cyber operations. Research conducted by this thesis extends that capability by enabling the development of sophisticated scenarios and planning for missions by defining a methodology for modeling and developing attacks against virtualized adversary networks. This research will benefit DoD cyber organizations in developing and practicing OCOs in a cost and time effective manner.

D. THESIS OUTLINE

1. Chapter II: Background

Chapter II describes the phases of a cyberspace operation in the context of the Joint Targeting Cycle. We describe how a target is selected and developed through cyber reconnaissance and information gathering. We describe common methods for collecting intelligence using active, passive and social engineering methods, and how vulnerability analysis is conducted using the gathered intelligence. This chapter also provides a brief overview of MAST and describes an example implementation.

2. Chapter III: Condition-Based Execution and Simulation

Chapter III defines our methodology for modeling cyberspace attacks. We define our model as a finite state machine and categorize the conditions under which an attack executes. We describe the model utilizing an example scenario and describe the current MAST configuration including its modules and architecture.

3. Chapter IV: Implementation and Testing

Chapter IV describes an example scenario used to implement our attack model. We provide a brief background and then step through the development of an attack scenario from the development of an attack to full testing of our model within MAST.

4. Chapter V: Conclusion and Future Work

Chapter V provides a summary of the research conducted in this thesis, its results and conclusions. We describe recommendations for future work to further improve the capability of MAST.

II. BACKGROUND

A. OFFENSIVE CYBERSPACE OPERATIONS

The information driven nature of modern military operations has created an ability and a demand to share information and communicate at distances and speeds that far exceed any capability in the physical domain. Cyberspace operations are an integral and interwoven aspect of DOD operations in the physical domains of air, land, maritime and space, and they are conducted to “retain freedom of maneuver in cyberspace in support of the commander’s objectives, and to deny freedom of action to adversaries” (Joint Chiefs of Staff 2013a, 7). Cyberspace operations are divided into three categories: Offensive Cyberspace Operations (OCO), Defensive Cyberspace Operations (DCO) and Department of Defense Information Network (DODIN) operations and defense. DCO and DODIN operations are critical to defending, securing and operating the DOD communication systems and networks. OCO are conducted to “project power by the application of force in and through the cyberspace” (Joint Chiefs of Staff 2013a, 8).

The cyberspace domain is defined as the “global domain consisting of the interdependent network of Information Technology (IT) infrastructure, Internet, telephone communication networks, computer systems, and embedded processors and controllers” (Joint Chiefs of Staff 2016, 58). Applying force in the cyberspace domain means achieving effects that jeopardize the confidentiality, integrity or availability (CIA) of an adversary’s systems or networks. Applying force through the cyberspace domain can be used to manipulate the CIA of a system in order to achieve an effect in the physical domain. An effect is a change in the physical or behavioral state of a target or target system, and is classified as desired and undesired. Desired effects help to achieve an objective, while undesired effects inhibit progress toward an objective or are generated as a result of unintended actions (Joint Chiefs of Staff 2013a).

B. JOINT TARGETING CYCLE

The application of force in the cyberspace domain is no different from the physical domains. Targets must be developed, validated, approved, de-conflicted, and effects must

support the commander's intent and be limited to the force necessary to achieve a military advantage. The Joint Targeting Cycle (JTC) is a six-phase comprehensive and logical process for employing the "ways and means to create desired effects" that support a military objective (Joint Chiefs of Staff 2013b, 11). The six phases of the JTC as defined by Joint Publication 3-60 are explained below.

1. Phase 1: The End State and Commander's Objectives

The first phase requires defining and understanding the mission, the commander's intent, and the desired end state. These help to establish the framework to identify targets that, if engaged, support the mission objectives, and required tasks that enable reaching the desired end state.

2. Phase 2: Target Development and Prioritization

Analysis, assessment and documentation help identify and characterize potential targets that, when engaged, support the achievement of the commander's objectives. This phase includes target system analysis that identifies the interactions between the physical, logical and social aspects of a system, and their relationship to other targets within the system. It is intended to identify critical systems and components, and their vulnerabilities.

3. Phase 3: Capabilities Analysis

Capabilities analysis is the process of evaluating all possible capabilities that can be used to achieve the desired effects against a target's critical elements. The purpose is to maximize the application of force while minimizing collateral damage, wasted resources and risk. This phase includes target vulnerability analysis, capability assignment, feasibility assessment, and effects estimation. Vulnerability analysis identifies components of a target that, if engaged, would reduce the target's ability to perform its designed function. Capability assignment aligns known vulnerabilities with an appropriate capability that can target a specific vulnerability to achieve the desired effect, also known as weaponeering. Feasibility assessment evaluates each assigned capability and its ability to exploit a target vulnerability. Effects estimation identifies the level of possible collateral

damage and potential diplomatic or public relations consequences of using a particular weapon or capability.

4. Phase 4: Commanders Decision and Force Assignment

The result of Capabilities Analysis is the identification of an appropriate capability to achieve the desired effect(s) on the target. During Phase 4, this capability is compared to available forces, sensors or weapon systems to allocate a specific asset to the target. Once allocated, the commander must approve the target and method of engagement in order for detailed planning to begin.

5. Phase 5: Mission Planning and Force Execution

Once the target has been approved by the commander, detailed planning and preparation is conducted prior to the execution of the mission. Planning and preparation include the development of courses of action, situation and intelligence updates, and mission rehearsals to increase the probability of success in reaching the Phase 5 end state, a successful engagement.

6. Phase 6: Target Assessment

Target assessment is a continuous phase that evaluates the results of the previous five phases of the JTC, and examines the effectiveness of the targeting process and target engagement. The effectiveness of an attack or engagement is determined through analysis of Battle Damage Assessment (BDA) and Collateral Damage Assessment (CDA), and may result in re-attack recommendations.

A cyberspace attack is an example of an OCO that aims to deny or disrupt an adversary's use of a cyber system, or manipulates that system and its information to generate an effect in the physical domain. Once a target has been identified that supports the commander's intentions, as stated in Phase 1 of the JTC, system and vulnerability analysis is conducted in Phases 2 and 3 to determine if a cyberspace attack is feasible. In order to do this, cyber reconnaissance is conducted to gather information about a specific target.

C. CYBER RECONNAISSANCE

There are three requirements for conducting a cyberspace attack: access to the target system, a vulnerability in the target system, and a payload that can be used against the target vulnerability (Denning and Denning 2010). Access can be physical, where an attacker can physically touch and interact with the system, or remote, where the attacker utilizes the Internet to interact with the system. Vulnerabilities are flaws in the hardware, software, or system-user interaction that allow an attacker to use the system in a manner other than originally designed. A payload is code – often malicious – that is executed on the target system to cause an intended effect. The goal of cyber reconnaissance is to discover as much about the target system as possible, in order to discover an access path and a vulnerability that will enable the delivery of the payload.

Vulnerabilities fall into three categories: system vulnerabilities, configuration vulnerabilities and user vulnerabilities. System vulnerabilities are weaknesses in the underlying hardware and/or operating system, installed software, or default settings that allow an attacker to interact or manipulate a system. Configuration vulnerabilities exist due to a failure to implement basic cyber defensive measures, such as implementing and correctly configuring firewalls, enforcing minimum password lengths, updating and patching identified vulnerabilities, or allowing non-administrators to change system configurations. User vulnerabilities are created by a lack of awareness of security risks, which can be attributed most often to a lack of training. These vulnerabilities are difficult to quantify, but are often the easiest to exploit using social engineering against an unwitting user. User vulnerabilities can be mitigated by implementing a security policy that follows the principle of least privilege, which ensures that users have access to only those resources that are necessary to do their job, to prevent a user from knowingly or unknowingly accessing critical system resources.

Vulnerabilities can also be classified as zero-day or non-zero-day. Zero-day vulnerabilities are hardware or software flaws that are not publicly known, and have no publicly released patch or update. Non-zero-day vulnerabilities are publicly known, and often have publicly available patches or updates. Zero-day vulnerabilities require significant time and resources to discover, but are the most valuable to an attacker because

they are the most difficult to detect and prevent. Despite the fact that non-zero-day vulnerabilities are publicly known, they are still present on networks and systems that have not been updated or patched, and provide an exploitable attack vector.

The following sections describe active and passive methods to gather intelligence and discover vulnerabilities in a target network. Active methods interact directly with the target and may provide more in-depth information, but also may provide indications and warning to the adversary of malicious activity. Passive methods do not interact directly with the target, but collect and analyze open source information about a target, and thus may produce more limited results. Both should be used for thorough and complete cyber reconnaissance.

1. Passive Collection Methods

Open source intelligence (OSINT) is any information that is available in the public domain. In the information age, it is almost impossible for a target organization to have no online presence; therefore, gathering OSINT from social media sites, public websites, and related news articles and publications establishes an initial footprint of a target. Directed OSINT searches can provide information about physical addresses, domains, email addresses, phone numbers, employee data, hours of operation, and job postings.

Many tools gather publicly available information through the Domain Name System (DNS). These tools are built into most operating systems, but are also accessible through the Internet. WHOIS is a DNS query tool that provides identifying information on registered domains and can also provide physical addresses, contact information for administrative and technical personnel, and DNS server information. Netcraft is a website that will scan a target domain and provide its DNS information, as well as services it is running, any known vulnerabilities, and a security assessment of the target (Engebretson 2013).

Many tools have been developed to automate the OSINT gathering process and develop profiles from intelligence that is collected. The Harvester is a python script that will search Google, Bing, and LinkedIn for information about emails, subdomains, and hosts that belong to a target address (Engebretson 2013). ThreatAgent Drone is a free

service that will search OSINT to develop a profile that includes IP address ranges, emails addresses, points of contact, and ports that are open by searching Shodan, which is another OSINT search engine for Internet-connected devices (Engebretson 2013).

Information gathered using OSINT methods can be compiled and analyzed to generate what is known as a target footprint. A footprint is a profile of a target's physical and cyber presence as well as its networks and security posture, and should include domain names, IP addresses, and potential email addresses. As OSINT methods tend to be passive, they should give the target no indication of a potential attack, however they do not provide as much information as system scanning and vulnerability analysis.

2. Active Collection Methods

Once a footprint has been generated with an IP address or range of addresses for the target, the next step would be to scan for hosts on those networks. The goal of scanning is to identify live hosts, as well as their operating systems (OS), applications and services running, and any vulnerabilities that may be used to exploit the target. Scanning a network is an example of an active reconnaissance method that may alert the target if not done correctly. There are tools available that will automatically scan a specific IP address or range of IP addresses, and detail host OS, open ports, and services running.

Ping is a system utility used to test reachability of a target by sending an Internet Control Message Protocol (ICMP) echo request. If the target IP is alive and not blocking ping requests or replies, it will reply to the sending machine. FPing is a tool built into most Linux distributions that uses ping to scan a range of IP addresses. FPing is essentially an automated version of ping that takes the burden off the user of sending pings to every host in an IP range. Echo replies from a target are recorded and identified as targets that can further scanned for vulnerabilities.

Nmap is an open-source utility that was designed for network discovery and security auditing (Engebretson 2013). Nmap scans a target IP for open ports and services running on those ports, and can determine the OS and protocols running on the scanned IP. It allows the user to determine the type of scan, the frequency of the scan, and the speed of the scan. Nmap options allow the scanner to scan in a covert manner, and while still an

active method, it reduces the ability of an Intrusion Detection System (IDS) to identify the scan as cyber reconnaissance. The decoy scan option puts the scanning IP in the middle of decoy IPs, making it much more difficult for the target to discern the IP that is actually scanning it. The Nmap time option allows the user to determine the pace of a scan, ranging from .3 seconds to five minutes between packets. This option is intended to prevent an IDS from recognizing a scan as suspicious behavior. The results of an Nmap scan can provide a good estimate of the OS running and open ports and services on a target, making it possible to identify potential vulnerabilities that are known to a specific OS.

Using the intelligence collected from OSINT, host discovery with Fping and port scanning with Nmap, it is possible to remain relatively covert up to this point, depending on type and frequency of scans. Further packet capture methods can be used to passively capture traffic on a network, providing valuable information to build a target profile.

A network vulnerability scanner is an example of an active method of cyber reconnaissance that will provide the most information about a target, but will also generate significant traffic and red flags on any IDS running on the network. Nessus is a well-known vulnerability scanner that provides free, limited licenses for home users (Kennedy et al. 2011). Nessus scans a target for vulnerabilities, potentially identifying the OS and specific software installed on the target. Nessus can only identify known, non-zero-day vulnerabilities that have been discovered and disclosed by security researchers or OS developers. Once vulnerabilities are disclosed, they are considered “dead” since patches or updates are issued to correct the vulnerability. At this point, they can only be exploited on unpatched systems (Engebretson 2013). Nessus scans with such detail that it can identify a system’s OS and patch version to determine if “dead” vulnerabilities are still present on the target. Nessus is a powerful tool for cyber reconnaissance, but must be used with caution to prevent detection and the loss of surprise.

3. Social Engineering

The methods described above were focused on gathering information on the target system and the network in which it resides. In an environment where computer systems are increasingly hardened against traditional attack vectors, one of the largest vulnerabilities is

often the user of that system. Social engineering is the art of exploiting human weaknesses and user interactions in the system. OSINT information gathering often produces employee lists, email addresses and job titles, which can be exploited and used to develop password lists to match email addresses discovered. Phishing emails or phone calls impersonating legitimate employees or IT support are common social engineering tactics, but the possibilities are endless.

D. TARGET SYSTEM AND VULNERABILITY ANALYSIS

Target system analysis is the process of evaluating physical, logical and social systems, and the interactions among them. Using the information that was gathered during cyber reconnaissance we build a complete profile of the target that includes the physical systems and how they are connected, the logical systems of network topology, and software and services that are in use on the target hosts, and finally the social systems of users and target environment. Once the profile has been developed, the total system can be analyzed for vulnerabilities.

1. Network Topology

A network topology is a graphical representation of the “layout and configuration of networked devices and the connections between them” (Collier 2016, 6). At a minimum, it should include all discovered routers, switches, and hubs that facilitate the routing of traffic, and all client devices that are connected such as computers, servers, printers, and wireless devices (Collier 2016). As new information is collected, it should be added to the topology, including specific host information, OS, services running, security posture and potential vulnerabilities. A complete network topology is key to visualizing the network and identifying potential access points and vulnerabilities. Figure 1 shows an example network topology with all connected devices in the target network, including routers and switches, network links to the Internet, and public facing services such as mail servers and web server. The identification of subnets is important as well, especially those that use network address translation (NAT) to prevent externally-generated traffic from entering the subnet. Although Figure 1 shows a rather complete network topology that could be generated from OSINT and use of scanning tools detailed above, in reality a complete

network topology is rare and inferences have to be made for gaps in information. Most network administrators and security policies will turn off echo replies from Ping requests and most firewalls will block externally-generated Transport Control Protocol (TCP) SYN requests for hosts that are not servers. There are other methods to discover live hosts on the network, including those that require establishing a presence on the network to conduct internal scanning, but as with any active method, these can create a lot of noise and the loss of any surprise if any actions are discovered. It is necessary to note any estimate in the topology for missing or inaccessible devices, even if they cannot be confirmed they must be planned for and considered when developing attack paths.

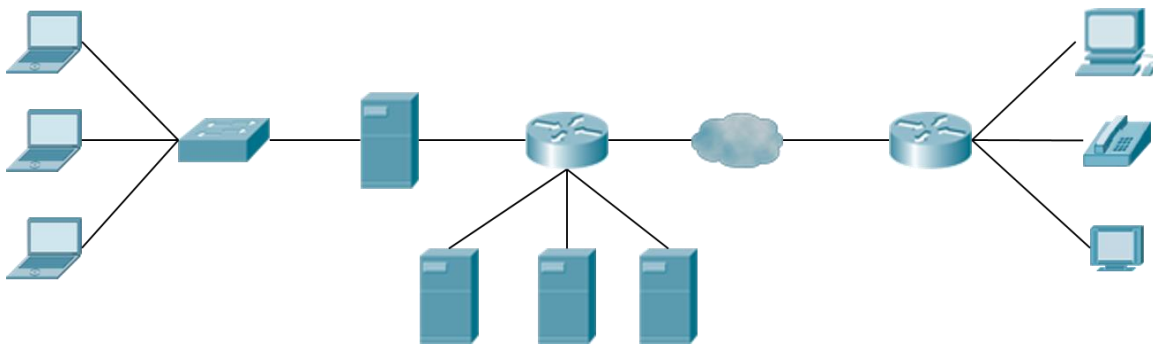


Figure 1. Example Network Topology

2. Enumeration

Enumeration of individual hosts on the target network provides usernames, user groups, hostnames, domains, IP routing tables, services running, as well as OS and application fingerprinting of a target. This enumeration data can be used to access or confirm active hosts within the system and to identify other hosts to pivot to in order to reach the target. There are multiple ways to enumerate a system, but it is important to note that full enumeration requires establishing a connection with the target host.

3. Host Physical Environment and Purpose

The physical environment where the target resides is an important aspect to understand what other type of vulnerabilities it is susceptible to. When examining the physical environment, if possible it is important to note the following characteristics:

- Typical user workload and schedule
- Facility hours of operation
- Temperature Sensitivity
- Physical Security
- External Devices (mouse, keyboard, webcam, printers)
- Relation to other workstations

The target environment and purpose are useful for developing an attack path that may not use the network directly to access the system, such as through an insider, or remotely through an external device with malicious code. Awareness of the physical environment can also indicate optimal times to execute an attack, such as at night or when a user is at lunch. The typical workload can provide information as to which software packages are installed, which applications are used the most, and insight into the user's vulnerability to attack or level of cyber security awareness.

4. Attack Graph

When the topology has been completed, which includes both inferred and confirmed connections and devices, vulnerabilities are overlaid to associate hosts, vulnerabilities and connections into a path. Combining all possible paths and their vulnerabilities results in an attack graph. Formally, an attack graph is a “succinct representation of all paths through a system that end in a state where an attacker has successfully achieved a goal” (Jha et al. 2002, 1). Attack graphs are used by Red Teams and analysts to evaluate the total vulnerability of a network that is the result of local and isolated vulnerabilities that can be exploited in sequence to further enumerate a system.

They are also used by network defenders to determine optimal placement of IDS to protect their networks. Attack graphs can have more than one starting state if there are multiple exploitable paths to access the system, and paths can cross or lead to the same state. Research by Jajodia, Sushil and Noel (2010) demonstrated the development of an attack graph as a model for gaining superuser privileges on a target's mail server by exploiting the target's webserver (see Figure 2).

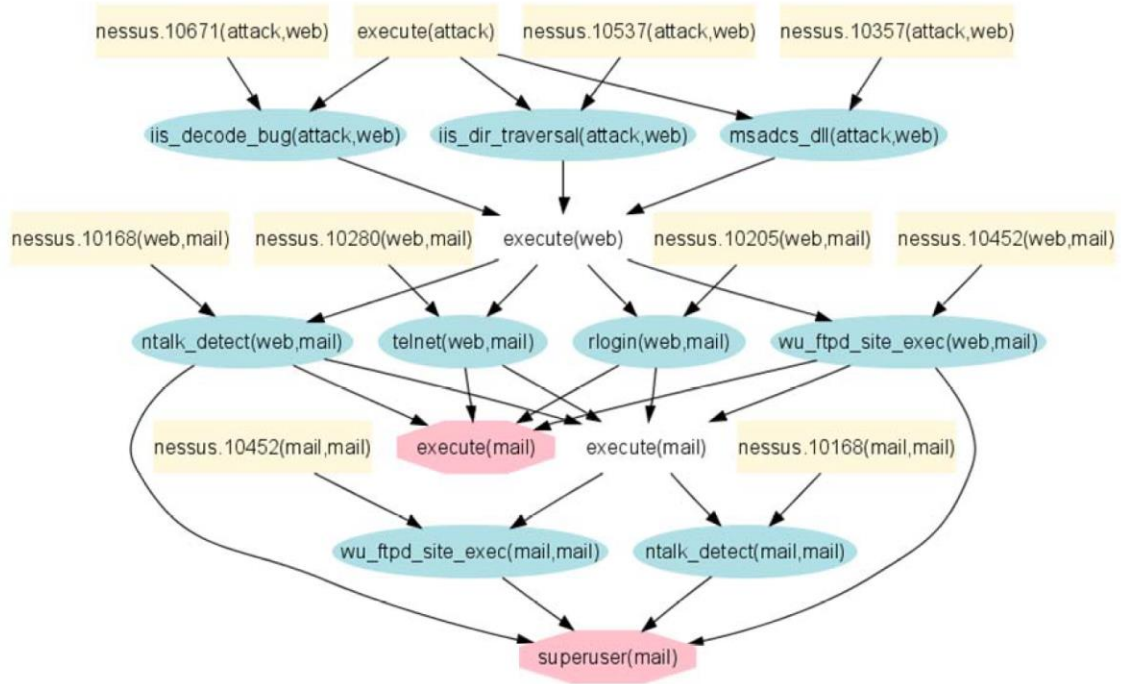


Figure 2. Attack Graph for a Test Network. Source: Jajodia, Sushil and Noel (2010).

The yellow boxes represent pre-conditions, blue ovals are exploits and red octagons are the desired end state. In order to utilize the blue exploits, the yellow pre-conditions must be met. As an example, the *iis_decode_bug(attack,web)* exploit requires the web server to have the vulnerability 10671 identified by Nessus, and the attacker must be able to connect to the webserver. When the pre-conditions are met, the payload is executed and the white box, a post-condition is achieved, in this case the ability to execute code from the web server. The white *execute(web)* is now a pre-condition for four additional exploits in

the path. The *ntalk_detect(web,mail)* is the shortest exploit path to the end state, and both pre-conditions are met as the attacker can *execute(web)* and the vulnerability 10168 has been identified to exploit. Just with this simple example, it is apparent that there are multiple paths to reaching the end state, this may not always be the case, but it provides the attackers with options to determine the right capability to reach the end state.

E. MALICIOUS ACTIVITY SIMULATION TOOL

The development of Malicious Activity Simulation Tool (MAST) began at NPS in 2011 as a means for training and testing network administrators on the very network they are assigned to manage. Originally known as Malware Mimics, it was intended to simulate the actions of a Red Team for network testing, without requiring an actual Red Team or malware. The framework was designed under the following premises (Taff and Salevski 2011):

- It must be safe enough to use on operational networks, and it must be inherently benign, controllable, and possess a failsafe for rapid neutralization.
- It must simulate the behaviors of malware without introducing actual malware into the system.
- It must be distributed, allowing a trainer to be geographically distinct from the test network and its trainees.

MAST has undergone significant testing and enhancements since it was first developed, but the basic framework and design have remained the same.

1. MAST Architecture

MAST is a three-tiered client server application, shown in Figure 3, that consists of a top-tier Scenario Generation Server (SGS), mid-tier Scenario Execution Servers (SES), and bottom-tier MAST Clients (Aybar 2017). At the top-tier, the SGS is responsible for overall control of the SES and MAST Clients; it maintains all training scenarios and can run multiple independent training scenarios simultaneously. At the mid-tier, the SES

receives commands from the SGS and facilitates the training scenario for the MAST Clients. The SES distributes scenarios to the clients, and controls their execution through the clients; it also maintains the kill-switch to immediately cease a training scenario. At the bottom-tier, the MAST Clients respond to commands from the SES, and execute the simulated malware (SimWare) modules that are installed to simulate attack effects (Aybar 2017). SimWare modules mimic the behavior of malware without causing any harm to the network or host. Currently MAST supports the following malicious behaviors: pings, scanning, browser hijacking, logic bombs, drive-by downloads, attack when idle, and targeting of a specific host. Following the principles of the original design of MAST, SimWare modules are not actual malware, but simulate the behavior of malware through benign methods.

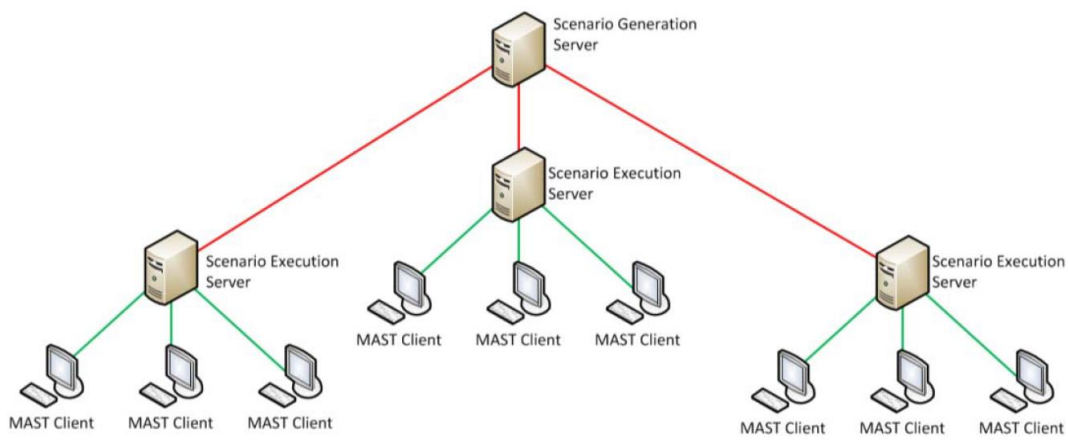


Figure 3. Three-Tiered MAST Architecture. Source: Aybar (2017).

2. Example MAST Scenario

An example implementation of MAST used to test network administrators during pre-deployment training was developed by Justin Neff, and is described below and in Figure 4 (Neff 2012):

- The training team, located in Ft Meade, Maryland issues commands from the SGS to the trainee SES to simulate a worm on the shipboard network.

- The SES, located aboard the training ship, directs the Clients to simulate scanning behavior to a set number of hosts.
- The Clients, installed on hosts on the trainee ships network, simulate the scanning behavior using SimWare modules for port scanning
- The SES directs additional Clients to exhibit the scanning to simulate the propagation of the worm.
- The network administrators identify the scanning behavior and take appropriate action to stop the propagation.
- Once appropriate actions have been taken, the training team ends the exercise.

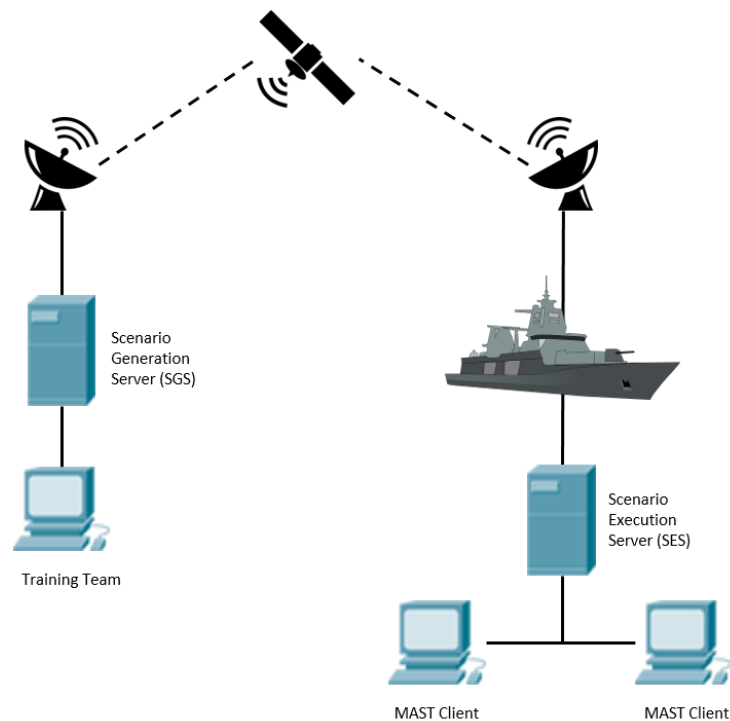


Figure 4. Example MAST Implementation

MAST has undergone significant enhancements, for example, its communication channels have been secured, and a scenario scripting language was developed to incorporate multiple SimWare modules into a scenario. As well, it was approved for shipboard testing. Most recently, three new SimWare modules were developed to demonstrate MAST's utility as a framework for developing and testing cyberspace attack scenarios (Aybar 2017).

3. Simulating Cyberspace Operations

As described, successful cyber reconnaissance should generate a target network topology, along with an attack graph that represents the possible paths to a successful attack. The modularity of MAST provides an environment where scenarios can be built for each attack path, using modules to test for conditions and SimWare modules to simulate the effect on the target, without damaging the host system. This allows an attacker to develop and test the sequencing on an attack from start to finish, and run the test against all possible attack paths to determine its effectiveness.

F. CHAPTER SUMMARY

This chapter provides an overview of offensive cyberspace operations, including the requirements and preparation necessary to execute an OCO mission successfully. It describes the processes and tools available to conduct cyber reconnaissance, target system analysis, and vulnerability analysis in order to analyze all possible ways to attack a target system. It further provides an overview of MAST and describes its utility as a tool for testing cyber weapons. The next chapter describes condition-based execution and simulation of a cyberspace attack.

THIS PAGE INTENTIONALLY LEFT BLANK

III. CONDITION-BASED MALWARE EXECUTION AND SIMULATION

The previous chapter described how the MAST framework supports simulating the effects of an attack on a target system using SimWare modules and triggering those effects using pre-built modules. This chapter explains how MAST can be used to simulate an entire attack path. We describe how we model the attack path and decompose it into its basic elements in order to effectively simulate all steps required to attack a target system.

For malware execution, an attack graph is used as a graphical representation of a finite series of attack states, and transitions between those states when certain conditions have been met. Linking successive states together forms an attack path, where each state represents a phase of the attack that meets some objective. Objectives are atomic events that must occur in the correct sequence in order for the attack to be successful. The objective may be the desired cyber effect, or an intermediary action or condition that enables a follow-on state. We define three categories of states for our model: the initial attack state, intermediary states and the end state.

The initial attack state is the first state in an attack path, and is reached when the initial conditions for the attack have been satisfied; the initial conditions are described later in this chapter. An attack graph may have multiple initial attack states when there are multiple attack vectors for targeting a network, but each path will have only one initial attack state. Intermediary states are all states the attack path passes through between the initial attack state and the end state. The end state is the final state in a path, where the attack has achieved the desired effect on the target system. Since attack paths often span multiple hosts across a network, each state consists of an identifier that links the state to a specific host on the physical network.

A state transition is triggered when a condition for a subsequent state is realized in the current state, an action is initiated, and the attack transitions to the resultant state. An action is some event that occurs on the target machine, the action may be the desired cyber effect, or it may be null if no cyber effect is desired, which is often the case with intermediary states. The transition function takes as input the current state, a condition to

monitor, and returns the state that the attack will transition to and the action that occurs when the condition is satisfied. If no condition is met, the attack remains in the current state. We define three categories of conditions: user conditions, system conditions and attacker-defined conditions.

Thorough and successful cyber reconnaissance and analysis may reveal multiple attack paths to reach the same end state, and within an attack path, a state may have multiple transitions to successor states. We represent a state with n transitions as a part of n separate attack paths where each state can only transition to one other state. We define an attack path as deterministic since the attack can only transition to one state when its condition is met. An example state machine generated from an attack graph is shown in Figure 5. State 1 is the start state, State 5 is the end state, and States 2, 3 and 4 are intermediary states. The conditions for state transitions are represented by the letters between states. There are 2 separate and unique attack paths that can reach the end state in this example. Since State 1 has two unique transitions, we identify the two attack paths as Path1 through State 1, State 2, and State 5, and Path2 through State 1, State 3, State 4 and State 5. These two paths combined represent the attack graph for the system.

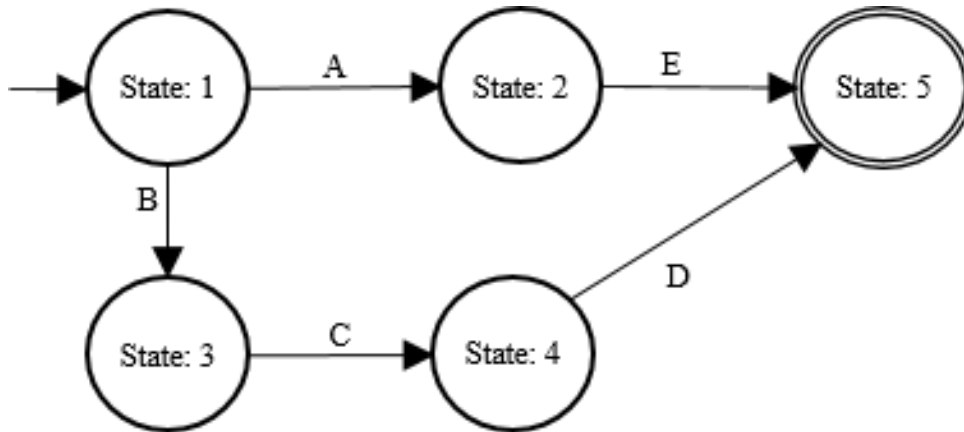


Figure 5. Finite State Machine

Identifying two unique paths in the example diagram, we would build two separate scenarios to simulate each attack path. In order for the scenario to transition from state to state, we need to define the conditions that will trigger the transitions. The following

sections describe the initial conditions required by the start state, the set of conditions for state transitions, and events that can occur once a desired state has been reached in order to detail the execution of an attack path.

A. INITIAL CONDITIONS

Chapter II defined the three requirements of a cyberspace attack as access to the system, a vulnerability in the system, and a payload that is executed to achieve a desired effect. In relation to the sequence of a cyberspace attack, these requirements translate to the initial conditions that are the conditions for the initial attack state, which must be satisfied before the attack can begin. It is important to note that these are conditions for a cyberspace attack, but they are not required for all cyberspace actions, such as cyberspace defense, or cyberspace intelligence, surveillance, and reconnaissance (ISR). The state machine for the initial conditions is shown in Figure 6, where each state is completed in a logical order as determined by the attacker.

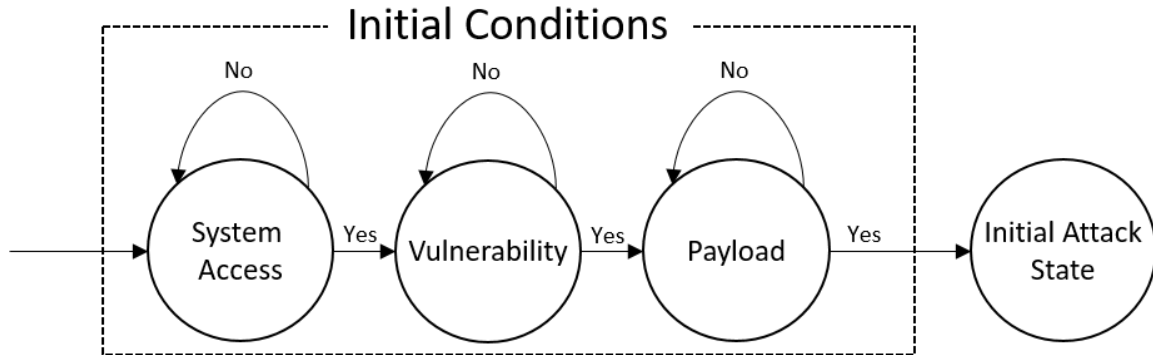


Figure 6. Initial Condition State Machine

1. System Access

In order to conduct a cyberspace attack, the attacker must have a method of accessing the system. Access is either physical, where the attacker or a third party can physically touch the device, or remote, where the attacker interacts with the system through the Internet. Neither physical nor remote access is trivial, as most modern networks and systems are hardened to prevent unauthorized access from either vector. The defensive

posture of any target is variable based on the level of perceived risk, and targets typically implement increased security controls to mitigate vulnerabilities, threats and the impact of any successful attacks.

Basic computer network defense (CND) controls that are implemented to prevent unauthorized remote access in a secure network include Firewalls, Demilitarized Zones, Honeypots, Network Intrusion Detection System, Network Intrusion Prevention Systems, and Network Segmentation. Physical security controls secure the environment where the target is located, and include restricted or controlled building access, fences, and guards. Logical controls such as password protection, biometrics, disk encryption, and secure boot are implemented to prevent an unauthorized user from accessing the device.

Initially, remote access does not imply anything more than that the target system is accessible via the Internet, either directly or on a network where other devices are accessible via the Internet. When a target is not directly connected to the Internet, a connection must exist between the target and Internet-connected devices on the same network that permit data transmission to the target. While this research focuses on targets connected by a local area network (LAN), there are other physical layer technologies that support data transmission including wireless local area networks (WLAN), Bluetooth and near field communication (NFC) that can be utilized to access a system not otherwise connected to the Internet.

2. Vulnerability

Once the method of accessing the target has been determined, identification of a vulnerability that can be exploited is necessary to launch a payload on the system. Chapter II described the categories of vulnerabilities that may be exploited on a system, including hardware, software, configurations, and even the users who access the system. We also described methods for discovering vulnerabilities using commercial vulnerability scanners such as Nessus and NeXpose, and the advantages and disadvantages of using these products. We differentiated between zero-day and non-zero-day vulnerabilities, and the limitation of most commercial vulnerability scanners to only identify non-zero-day vulnerabilities.

When targeting complex networks, it is possible to identify numerous vulnerabilities that provide access to the target host or network. But the discovery of these vulnerabilities alone is only half the battle. An exploit must exist to take advantage of the vulnerability, or if not already in existence, the exploit must be developed. The Metasploit Framework (MSF) is an application that is used by penetration testers to test the effectiveness of a network's defenses (Kennedy et al 2011). MSF provides a searchable database of known exploits that can be used against the vulnerabilities discovered in the Nessus or NeXpose scan (Kennedy et al 2011). One of the most common types of exploits is a buffer overflow, where a program fails to check or validate the size or type of input, and the attacker injects code that redirects the execution of the program to a location that he or she desires. The execution of the program may be redirected to a memory location that contains code the attacker wants to execute; the code that the attacker wants to execute is known as the payload.

3. Payload

While not lessening the importance of system access and a vulnerability with ability to exploit it, a payload is the essential element of cyberspace attack. In the physical domain, a precision strike weapon has two elements: some form of ordnance and a delivery vehicle that transports the ordnance from the launch point to the target. In the cyberspace domain, the payload is the ordnance that executes to achieve the desired effect on the target, and access and vulnerability combined are the vehicle that delivers the payload. The type of payload is dependent on numerous requirements, e.g., it is designed based on the effects that are desired, and within constraints that are determined by the configuration and security measures implemented by the target system. Examples of payloads are:

- **Data Exfiltration:** payloads that transfer data from the target system back to the attacker or some intermediary. Data exfiltration can be targeted, where a specific file or type of file is desired such as plans or intelligence, or sweeping for any information.
- **Backdoors:** payloads that enable accessing the target system without the use of an exploit after initially gaining access to the system.

- Shells: also known as terminals or command prompts, these payloads give attackers access to the target as if they were actually on the machine itself through a command shell. Two types of shells are bind shells and reverse shells. If a bind shell is used, the attacker is able to connect directly to the shell through an inbound connection, meaning a firewall does not prevent inbound connections. Reverse shells require the attacker to establish a listener that is waiting for an inbound connection, and then the payload must call back to the listener to establish the connection. Reverse shells are useful when firewall rules filter inbound connection requests, but do not filter outbound traffic.
- Key loggers: payloads that record a user's keystrokes on an input device, such as a keyboard or a touchscreen. These payloads are often used to record login credentials that can be used for further exploitation.
- Disk Encryption: payloads that encrypt the contents of memory on the target. These prevent reading of data on the device, unless the decryption key is provided. Disk encryption can be used to deny access to a device, and has been used recently to hold data ransom by the attacker.
- Denial of Service: payloads that deny a user access to information or a services. These payloads typically overwhelm a target system resources to the point where they crash or severely degrade that system's performance.

The initial conditions described above are the minimum required to conduct a cyberspace attack. Although the state machine in Figure 6 implies that the initial conditions must be satisfied in sequential order, in reality the order will depend on the mission and on the attacker. An attack may depend on using a specific payload to achieve the desired effect, in which case access and finding a vulnerability to deliver the payload will be satisfied second and third, respectively.

Satisfying the initial conditions allows an attack to enter the initial attack state and begin the sequence of testing for the conditions that will enable the attack to reach its end

state and achieve the desired effect on the target machine. The following sections will describe how we define conditions and how they are developed through an example attack scenario.

B. CONDITIONS

A transition between two states in the attack path is triggered when a specific condition for the successor state has been satisfied in the present state. One can also consider a state to be a phase in the attack process, where some objective (an atomic prerequisite event) has been met in order to reach that state. We define the condition for a state to be a test of whether or not the objective for that state has been achieved in the current state. When the condition is achieved, the attack transitions to the next state. If the next state is the end state, the attack path is complete, and thus the cyber effect is realized on the target. If the next state is an intermediary state, that state's transition function tests its condition to determine whether the attack should continue along its path. We use the following scenario to further explain this terminology:

- Alice wants to attack Bob by installing a malicious file to exfiltrate data when Bob is not at his computer.
- Alice determines that Bob is using an unpatched, vulnerable web browser.
- Alice will use a phishing email to coerce Bob to visit a malicious webserver she has set up that will exploit Bob's browser vulnerability to download her malicious file.
- Alice determines that the following events must occur on Bob's computer in order for her attack to be successful:
 - A: Bob must be logged onto his computer.
 - B: Bob's computer must have the vulnerable version of the browser installed.

- C: Bob must click the link in the phishing email to launch the vulnerable browser.
- D: Bob's browser must connect to Alice's webserver.
- E: Alice's malicious file must be downloaded by her webserver to Bob's computer.
- F: Alice's malicious file must detect when Bob is not at his computer.

The network topology that was generated from Alice's reconnaissance is fairly simple, Figure 7 shows how Alice is able to access Bob's computer directly from the Internet and does not have to access any other devices on Bob's network. In this example, Alice has determined her attack vector and identified the vulnerability that she wants to exploit, effectively choosing her attack path. The attack model for Alice's attack path against Bob's machine, shown in Figure 8, and corresponding transition table, shown in Table 1, detail the conditions that must be met in order for each transition of the attack path to take place.



Figure 7. Network Topology for Alice's Attack

When the attack is in State 1, the transition function is testing for the condition *User_Logged_On(Bob)* since the objective of State 2 is that Bob be logged onto the target system. *User_Logged_On()* is a function that identifies the user currently logged onto the system, and returns *True* if the username matches the argument passed to the function. Until *User_Logged_On(Bob)* returns *True*, the attack will remain in State 1. When *User_Logged_On(Bob)* returns *True*, the attack transitions to State 2, where the transition

function will now test for *App_Installed(VulnBrowser)*. *App_Installed()* is a function that returns *True* if the argument application is installed on the target system. If the application is not installed, or has been updated to a new version, the vulnerability that Alice planned to exploit might have been patched, and this attack path would no longer be valid.

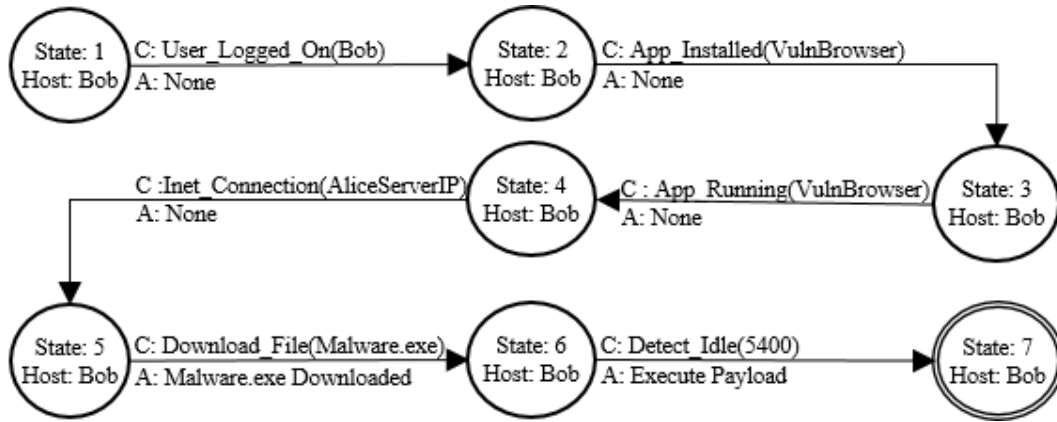


Figure 8. Attack Model for Alice's Attack Path

Once determined that the VulnBrowser is installed on Bob's computer, the attack transitions to State 3, where the transition function tests for the condition *App_Running(VulnBrowser)*, which returns *True* when the argument application is a process running on the target machine. This condition tests whether Bob actually opens the link with the VulnBrowser, as opposed to another browser that may not be vulnerable. Once the condition is met, the attack transitions to State 4. At this point, the transition function will test for the objective of State 5, which is that a connection has been established between the target computer and the web server set up by Alice. The function *Inet_Connection(IPaddress)* tests for an established connection between the target and the argument IP address, in this case the IP of Alice's web server. Once the connection is confirmed, the state condition has been met and the attack transitions to State 5.

In State 5 the transition function tests for the condition *Download_File(Malware.exe)*, which returns *True* when Alice's malware is downloaded from the webserver to Bob's computer. When *Download_File()* returns *True*, the attack transitions to State 6. The next transition *Detect_Idle(5400)* is used to determine when Bob

has not actively used his computer for 90 minutes (5400 seconds). The *Detect_Idle()* function checks for movement from the mouse or input via the keyboard to determine whether the device is currently in use by a user. After no input from either device has been detected for the argument time period, the function returns *True* and the attack transitions to State 7. In the state machine (Figure 8), State 7 is outlined with two solid lines to indicate that it is the end state. When an attack has reached the attack path end state, this indicates that it successfully achieved the desired effects on the target.

Table 1. Transition Table for Alice’s Attack Path

Current State	Condition	Action	Result State
1	User_Logged_On(Bob)	None	2
2	App_Installed(VulnBrowser)	None	3
3	App_Running(VulnBrowser)	None	4
4	Inet_Connection(AliceServerIP)	None	5
5	Download_File(Malware.exe)	Malware.exe Downloaded	6
6	Detect_Idle(5400)	Execute Payload	7

In order to focus the development of additional modules we define three categories of conditions: System, User, and Attacker-Defined, which are explained in the following sections.

1. System Conditions

System conditions include a system’s architecture and configuration. These allow an attack path to confirm the presence of a vulnerability, and to determine the effectiveness of a payload that is developed for a specific system. System conditions include, for example, identifying 64-bit versus 32-bit architectures, identifying a specific operating system installed and its version, identifying applications installed and their versions, and

any external devices attached to the device. Identification of Anti-Virus and IDS programs and their status are also system conditions.

System conditions also include the status of the system, such as CPU usage levels, memory availability, network status and connections, power and battery status, users logged on, and running processes and services. Similar to its architecture and configuration, the system's status is used to confirm the presence of certain conditions that the exploit and payload depend on.

2. User Conditions

User conditions are actions or events that are triggered by a specific user interaction with the system. User conditions include, for example, opening a specific program, visiting a specific website, downloading a file, or locking the computer. Social engineering tactics are also included within user conditions, such as convincing a user to change certain settings of a system, or to provide their password to an attacker posing as a system administrator. Lack of user interaction on the target computer is also a condition, as described in the earlier scenario.

3. Attacker-Defined Conditions

System and user conditions both depend on events or actions outside the control of the attacker. For example, there is no way to guarantee that a user will open a file, or that he will be idle for a specified time. In these cases, the attacker must define when a transition occurs according to chosen parameters. Attacker-defined conditions cover these cases, and might include triggering an event at a specific time, or after a specified amount of time passed, or based on a command from a server, or some other variable defined by the attacker.

In the scenario described above, we showed how conditions are developed from the atomic events that must occur during the execution of an attack, and how we link together the states and transitions to form an attack path. By testing atomic events individually and in sequence, we build simple functions that can be linked together to model complicated attack paths. This concept of attack path conditions can be seen in the development and execution of an actual cyberspace attack such as Stuxnet.

Stuxnet was a “large, complex piece of malware with many different components and functionalities” (Falliere, Murchu and Chen, 2011, 1). Stuxnet was packaged as a large dynamically linked library (.dll), that was divided into thirty functions, identified as exports, where each export had a specific purpose in controlling the execution and propagation of the attack. Export 15 was one of the first of these functions to run when Stuxnet was initially launched from a new host. The control flow for Stuxnet Export 15, shown in Figure 9, demonstrates how, starting with “Check CFG,” the function runs through condition checks before it can execute, and signal that conditions are set for the next export function. Similar to the state conditions that Alice’s attack had to meet in our example, the conditions for Export 15 must have been met before the attack would continue.

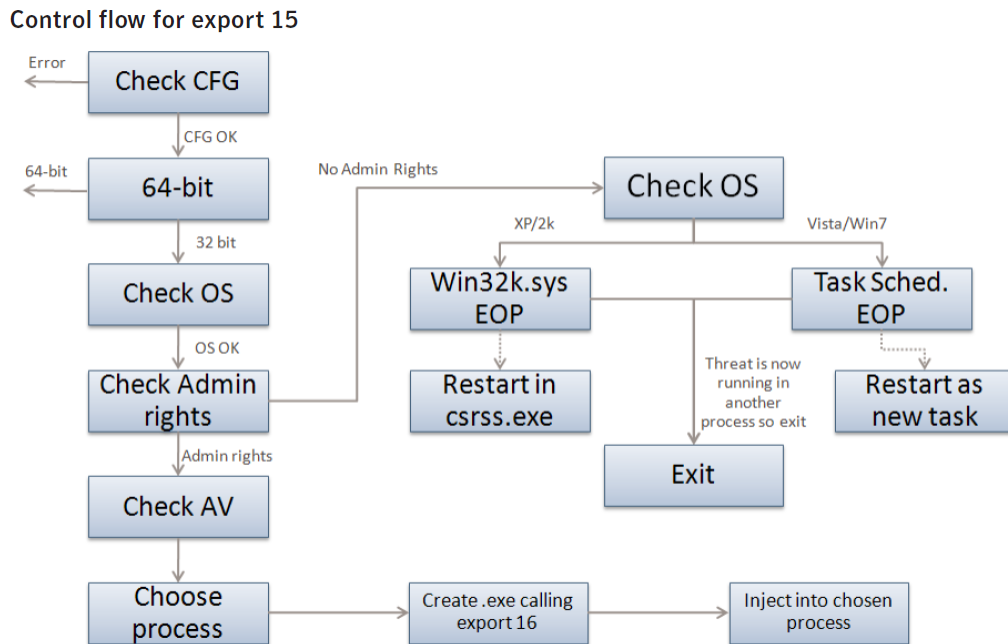


Figure 9. Stuxnet Export 15 Control Flow. Source: Falliere, Murchu, and Chen (2011).

The Export 15 conditions are all examples of system conditions in our model, in that they check for specific conditions that must be present within the system configuration or architecture before the export function can proceed to a new state. In most cases, if a specific system condition was not met, the function would exit to prevent detection. In the

case of “Check Admin Rights” testing false, the export would attempt to elevate its privileges using two zero-day vulnerabilities; if these were not successful, the function would exit (Falliere, Murchu and Chen, 2011).

The Stuxnet conditions provide a more granular and real-world example of conditions that should be satisfied within the execution of an attack path. Just as the conditions were satisfied during the actual attack, these conditions could also be tested and simulated within the MAST framework.

C. CYBER EFFECTS AND MAST SIMWARE

Earlier in the chapter, we described a payload as the ordnance that achieves the desired effect on the target machine in a cyberspace attack. The desired effect is generally one that violates the confidentiality, integrity, or availability of a target system - the three elements of the CIA Triad. Nieves, Dempsey, Pilliteri (2017) in their research as part of the National Institute of Standard and Technology (NIST) define confidentiality as “Preserving the authorized restrictions on information access and disclosure” (2). They define integrity as “Guarding against improper information modification or destruction and ensuring information non-repudiation and authenticity” (3). They define availability as “Ensuring time and reliable access to and use of information” (3). Payloads are designed to do explicitly the opposite of these definitions, i.e., they are designed to gain unauthorized access to information, to modify, delete, or destroy information, and to prevent access to information. In order to simulate payloads and their effects within the MAST framework for testing and rehearsal purposes we use SimWare, described below.

MAST was originally designed under the constraints that it must be safe to use on operational networks, and that it must be inherently benign while still mimicking the behaviors of malicious software. These constraints are met using SimWare to simulate the effects of malware while not endangering the network and system that execute the modules. Examples of SimWare modules in MAST include:

- **EICAR Test File:** The European Institute for Computer Antivirus Research (EICAR) developed the EICAR signature, which is a 68-character string used for testing Antivirus software. This module writes the EICAR string

to a target computer's disk, then the string is recognized by the installed antivirus as a virus being installed on the target machine.

- **DriveByDownload:** This module simulates a common exploit in web browsers, where the user is asked to download a file while browsing a website. If the user agrees to the download, the SimWare module will trigger the EICAR test file to simulate a virus being installed.
- **Network Traffic Generation:** This module integrates the Nmap utility to generate network traffic to simulate a system infected with a worm. The module uses Nmap to generate scanning traffic similar to a worm that scans for adjacent hosts to propagate to.

D. CURRENT MAST DESIGN AND METHODOLOGY

MAST provides a framework where an attacker can develop an attack using the available SimWare modules, or develop their own modules and link them together to simulate an attack. There are two types of modules that are available in MAST: SimWare modules such as the examples described in the previous section, and modules that test for conditions. The condition modules currently available in MAST are:

- **TargetSpecificHost:** This condition returns *True* when a specific IP or MAC address is reachable by the attacker.
- **LogicBomb:** This condition returns *True* when the system clock on the target machine matches the time provided by the attacker.
- **AttackWhenIdle:** This condition returns *True* when the mouse and keyboard on the target machine have not been used for a specified number of seconds.

Previous research and development have shown that the condition modules currently available in MAST can be used in conjunction with the SimWare modules to simulate an attack on a target system. Most recently, Aybar (2017) showed how MAST was used to successfully simulate attacks against a single host based on temporal and logic-based conditions, using LogicBomb and AttackWhenIdle modules. The goal of this

research is to expand the capability of MAST to simulate the attack path model that was defined earlier in the chapter.

Earlier work described by Aybar (2017) details scripting language that was developed for MAST that “allows for the incorporation of multiple modules into a scenario file that can be customized, saved and run on any instance of a MAST client” (14).

E. CHAPTER SUMMARY

In this chapter, we described our model of cyberspace attack paths, and how we break down those paths into their basic elements of states and transitions. We also defined the requirements for transitioning between states, and how those transitions are developed from atomic events that must occur in order for the attack to be successful.

We described the initial conditions for an attack, and provided an example scenario for determining the states, transitions, and conditions needed for those transitions within the scenario. We further described our categories of conditions, including user, system, and attacker-defined. We finished with an overview of MAST, its current SimWare modules, and a discussion of the ability to combine modules into a scenario with the scenario scripting capability.

Chapter IV describes the implementation of our attack model to simulate an attack path on a simulated enemy network.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION AND TESTING

This chapter describes the implementation and testing of our attack model in a virtualized environment utilizing the MAST framework. We detail our development and implementation through an example scenario that demonstrates how we identify our attack path, and decompose the path into its states and transitions that comprise the attack model. We also describe the MAST modules that were developed to support this attack. We then execute our attack model against a virtualized target network, simulating the entire attack path from the initial attack state to the end state.

Our methodology parallels the development of a real-world cyberspace attack that we simulate with MAST to determine its effectiveness. Our test scenario utilizes a notional target within a controlled environment; as such, the scenario simulation begins after reconnaissance and vulnerability analysis of the target are complete, and any information normally gathered during these phases is also notional and provided to supplement the scenario.

A. IMPLEMENTATION

1. Scenario

In our test scenario, the target is an adversary command and control (C2) network that is used to oversee and control the physical security measures in use to detect and prevent unauthorized access to the adversary's base. The users of the network monitor video and sensor feeds that provide indications and warnings in order to alert security forces of any incoming attacks. The base has been identified as the location of a raid to secure a high value target. In order to enable the operation, cyber planners have been tasked with conducting a cyberspace attack to disrupt the target's C2 network in coordination with the raid to prevent early warnings and maintain the element of surprise for the ground forces. Cyberspace actions including information gathering and reconnaissance may begin as needed, but the effects of the attack must not occur until the time specified by the ground forces.

Reconnaissance of the target using open source, passive and active methods resulted in determination of the network topography shown in Figure 10. The topography shows five active host within the 10.0.1.0/24 subnet, identified as the C2 network. The C2 network is connected to the 10.0.2.0/24 subnet, which we identify as the DMZ. It hosts the adversary network's domain controller, DNS server, and mail server.

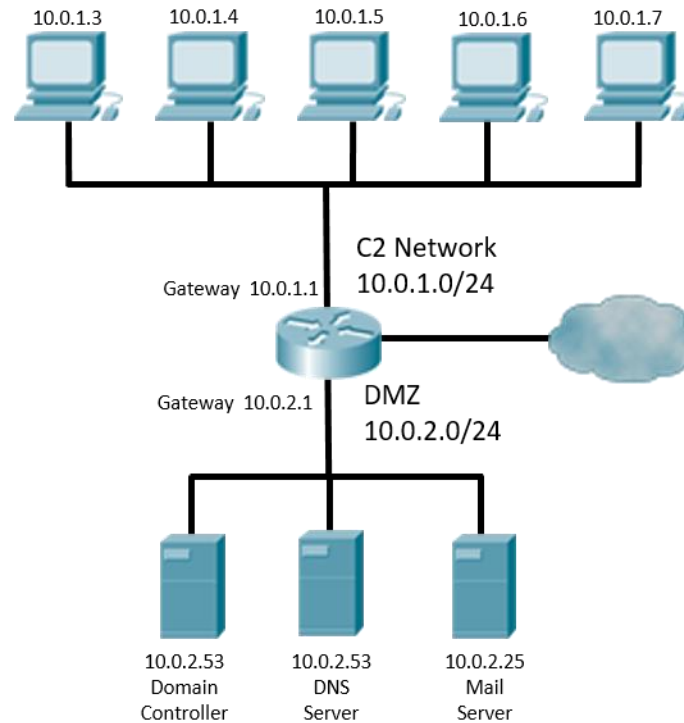


Figure 10. Target Network Topology

The network topography also shows that the adversary's local area network is connected to the Internet, which may make the target remotely accessible. Access to the target may be prevented by any firewalls or access control lists that deny externally initiated traffic from entering the C2 network, which may limit our attack vectors but indicates potential access nonetheless.

Further scanning of the network was conducted to fingerprint the operating systems running on the target hosts and to identify any services or open ports that may be vulnerable

to attack. Table 2 shows that all of the hosts within the C2 network are running Windows 7 Service Pack 1 (SP1) and the servers within the DMZ are running Windows Server 2012. The ports that are open and services that are running provide indications of the purpose of each host. The host with IP address 10.0.2.53 has port 53 open and DNS running, as well as port 139 and the Lightweight Directory Access Protocol (LDAP), which indicates this host is both a DNS server and the domain controller. The target with IP address 10.0.2.25 is assumed to be the mail server since it has port 25 open and is running SMTP and IMAP.

Table 2. Operating Systems, Ports and Services

IP Address	Operating System	Ports	Services
10.0.1.3	Windows 7 SP1	445	MSRPC
10.0.1.4	Windows 7 SP1	445	MSRPC
10.0.1.5	Windows 7 SP1	445	MSRPC
10.0.1.6	Windows 7 SP1	445	MSRPC
10.0.1.7	Windows 7 SP1	445	MSRPC
10.0.2.53	Windows Server 2012	53, 88, 135, 139, 389	DNS, MSRPC, LDAP, SSL
10.0.2.25	Windows Server 2012	25, 80, 81, 135, 139, 443, 444, 587	SMTP, HTTP, IMAP, MSRPC, SSL

Identification of the host operating system provides a baseline of default installed programs, settings, configurations, and possible default accounts and passwords. Having identified the operating systems, ports, and services that are running on each of the target hosts, we can then identify known vulnerabilities associated with each host and generate an attack graph to identify potential attack paths for our target.

2. Attack Graph

Chapter II described how attack graphs are developed by overlaying the vulnerabilities associated with each host in the target network and linking those vulnerabilities together to create paths that can reach the specified target. Chapter II also described how vulnerabilities are identified through scanners such Nessus and NeXpose,

which actively scan and probe the target and provide reports of known vulnerabilities that may be present on the system. We also note that these scanners are considered “loud,” meaning that the scans will most likely be identified by any network intrusion detection system (NIDS) that is running on the target network. This scanning would provide indications and warnings to the adversary that they are being actively scanned, possibly to gather information for an attack. Knowledge of the operating system installed on the host also provides a good method of identifying vulnerabilities without actively scanning the system. Websites such as www.cvedetails.com and Microsoft’s security tech center provide bulletins of known vulnerabilities and security patches and updates for these vulnerabilities, but many systems are not patched, leaving them vulnerable to attacks.

Chapter III described the necessity of identifying an exploit that can take advantage of the vulnerability and the utility of the MSF exploit database for searching and identifying exploits. In order to prevent early detection of our attack by using active vulnerability scanners we developed the attack graph for our target network, as shown in Figure 11. This was developed based on the operating system fingerprinting and the common vulnerability and Exposure (CVE) database www.cvedetails.com, which also lists known exploits for the vulnerabilities.

Our attack graph consists of three attack paths - two remote and one client-side. The remote attack is one where the attacker interacts with the system through the Internet and requires no interaction from the user. The remote attack vector begins with the `ms14_60_sandworm` exploit on the mail server or the DNS server. The conditions for this exploit include remote access to the target, and that the CVE-2014-4114 vulnerability is present on the system (cvedetails.com, 2015). This vulnerability affects Windows Vista SP2, Windows 7, Windows 8, Windows Server 2008, and Windows Server 2012, and allows an attacker to execute arbitrary code on the system (cvedetails.com, 2015). The `ms14_060_sandworm` exploit takes advantage of this vulnerability and is present within the MSF exploit database. Using the exploit and vulnerability identified, we are able to execute code on the Mail or DNS server, which provides a foothold on the network and allows us to attack the network from within.

Using the same exploit and vulnerability, we then attack the C2 host and deliver our payload that will disrupt the C2 network and enable the ground operation. We attack the C2 network from the mail or DNS server, despite the fact that the externally initiated traffic was blocked by the C2 network firewalls, and we are able to access the mail server externally. This makes our traffic appear as if it is coming from mail server and is allowed through the firewall to the C2 network.

The client-side attack path also accounts for the firewall blocking external traffic into the C2 network but mitigates it by having the user initiate a connection to a webserver from outside of the C2 network. Once user initiates the connection, data from the web server is not blocked as long as the connection remains active. The client-side attack path begins by exploiting the user of the system with a phishing email that appears legitimate. When the user opens a link with the default browser installed on the system and initiates a connection to a server set up by attacker, it enables the attackers traffic to enter the C2 network. This then initiates a drive-by-download of the payload from the malicious server. Once the payload is downloaded, it executes at the time designated to enable to ground operation.

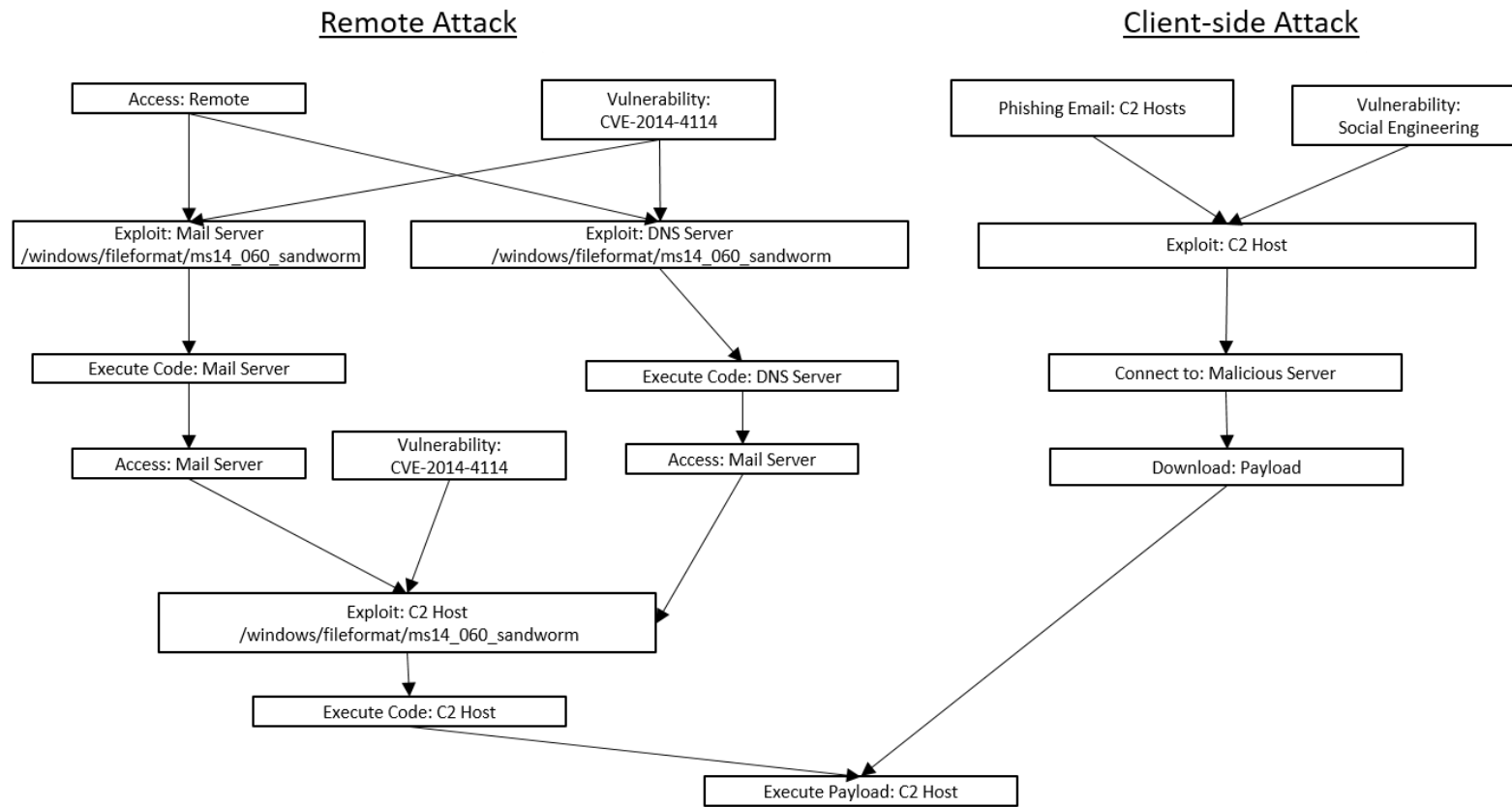


Figure 11. Target Network Attack Graph

The attack graph for our target network represents three attack paths, but it is not inclusive of all possible attack paths that may exist. Time and necessity will drive the development of additional and often more complicated attack paths, but for the purposes of our scenario, these three represent the distinct paths that will result in achieving the desired the effect on the target host. Since we did not conduct any active vulnerability analysis, it is difficult to confirm that the vulnerabilities are still present on the system. To mitigate this, we analyze the attack paths to determine which path is most likely to be successful. The remote attack paths utilize exploits and vulnerabilities that were released in 2014 and Microsoft also issued a software update to patch the vulnerability (docs.microsoft.com, 2014). Each of the remote attack paths also require two exploitations against two separate operating systems, which increases the potential for failure.

The client-side attack does not rely on a software exploit, but rather a social engineering exploit of the human vulnerability. By targeting all users on the adversary's network, we increase possibility that one will access the malicious server and initiate our attack. Comparing the three attack paths, based on probability of success we model and simulate the client-side attack to demonstrate its ability to achieve effects on the target.

3. Attack Model

Using the client-side attack path as our base we decompose the path into its atomic events and identify the states and transitions that will comprise the attack model that we can then simulate within the MAST framework. In this scenario, two of the three initial conditions required for the attack have already been met: access was identified as remote, as the payload is delivered by the remote server; and the vulnerability is the system user, who will be exploited using social engineering methods. The payload is the ordnance that achieves the objective, which is to disrupt the C2 network operations in order to enable the ground operation. The payload that we develop for this scenario will deny the users of the network access to their devices and prevent them from identifying any incoming attacks from the ground forces. Having satisfied the initial conditions for the attack, we develop the attack model, shown in Figure 12, that represents the events that must occur in order for the attack to be successful.

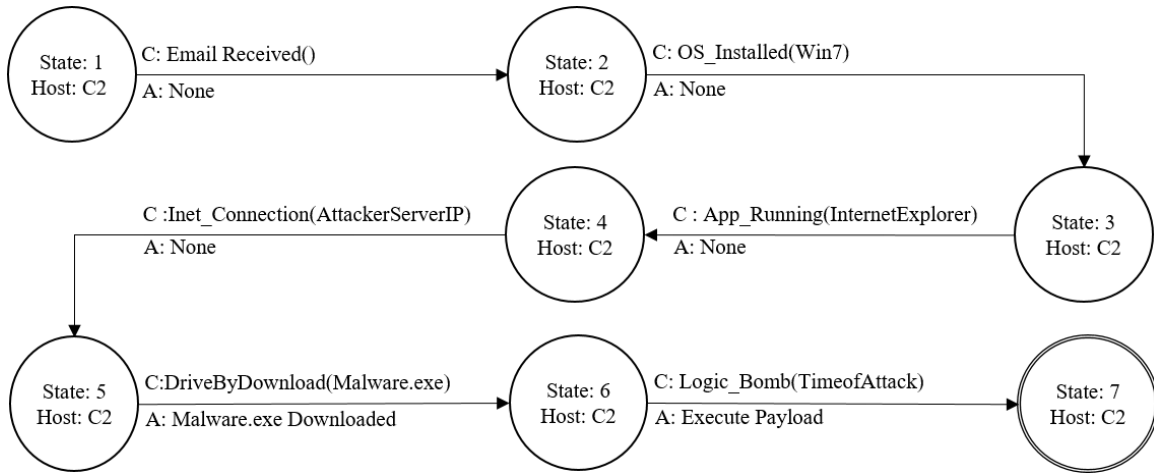


Figure 12. Attack Model for the Scenario Attack Path

The initial attack state, shown as State 1, is the state of the attack after the initial conditions have been met. The attack transitions when the phishing email is received by the users on the C2 network. The payload that we developed must be executed on a Windows 7 operating system, therefore, in order for the attack to continue we will verify the target is running Windows 7. Once the OS is verified, the attack transitions to State 3 where the transition function tests whether the user has clicked the link to launch a web browser, specifically Internet Explorer. When Internet Explorer is detected as a running process on the target, the attack transitions to State 4.

Our method of delivering the payload to the target is through a web server that the client connects to when they click the link in the phishing email. In order to transition to State 5, a connection must be detected between the target and the IP of our web server, once detected the attack transitions. When the attack is in State 5, we launch the drive-by-download that asks the user to download the file. If the user accepts the download, the action is to download the payload to the target and the attack transitions to State 6.

Once the payload has been downloaded to the system, there is no longer any user interaction required for the payload to execute. The objective of the attack is to achieve the desired effects at the time specified by the ground force to limit advance warning of the raid. In order to transition to State 7, the end state of the attack path, we test for a specific

system time. When that time is reached, the transition function triggers the action which is to execute the payload, and then transitions to State 7, the end state of the attack.

At the heart of the MAST framework are the modules that test for the conditions on the target host and simulate effects of the payloads with SimWare. Using our attack model, we develop modules to test for the conditions necessary to simulate our attack path.

4. Module Development

The MAST example scenario described in Chapter II explains how the MAST Client software is running on each host within the scenario and executes the commands issued by the SES. When a client connects to the SES, it identifies which modules it can execute. The modules are executable batch (.bat) scripts, shell (.sh) scripts, and Java archive (.jar) files that are launched by the client. Once executed, the modules return a value that is captured by the client and passed back to SES to determine the next command to be sent to the client. The client uses the Java Process Builder class to execute a module as sub-process, and the Process Builder method `getInputStream` to capture output from that process and return it back to SES. This design allows for modules to be written in any language, as long as the module can be executed, and return a value either by printing it to standard out or as an exit code. This value is interpreted by SES to determine the next event in the scenario.

Simulating the scenario attack model within MAST required the development of four modules: three modules that test for conditions, and one SimWare module that simulates the effects of our payload on the target. Combining these modules with the LogicBomb and DriveByDownload modules already available in MAST, we completed our attack model and can then simulate the complete attack path with MAST. The condition modules developed for this scenario are TargetSpecificOS, AppRunning, and InetConnection, and the SimWare module to simulate the desired effects on the target is called LockOut. An additional module, TargetUser was also developed but was not used in the example scenario.

TargetSpecificOS and TargetUser were both written in Java utilizing the Java Runtime Environment (JRE) 1.8.0 and compiled with the Java Development Kit (JDK) 1.8.

TargetSpecificOS requires two arguments: the name of the OS that is being targeted, and its version. The module returns 1 if the OS is detected, 0 if not, or 456 if an error occurs. The program is comprised of two classes, *TargetSpecificOS* and *VerifyOS*, which verify the input to the program and identify the OS installed on the host. The *TargetSpecificOS* class, verifies the input to the program for validity. If the input is valid, it initializes a new *VerifyOS* object that takes the two arguments and attempts to match the arguments to the OS installed. The *VerifyOS* class utilizes the Java class *System*, which provides access to system properties of the installed OS. *VerifyOS* utilizes the *System.getProperty()* method, with the argument *os.name*. *VerifyOS* compares the result from *System.getProperty("os.name")* with input to the program and returns the value depending on the result of the comparison. Appendix A contains the entire source code for TargetSpecificOS.

TargetUser is similar to TargetSpecificOS in that it has two classes and utilizes the Java class *System*. The program takes one or two arguments to account for user first and last names. The program's two classes are *TargetUser* and *VerifyUser*. *TargetUser* verifies the input to the program and provides feedback if the input is not correct. If the input includes a first and last name, it joins the two to create one string to pass to the *VerifyUser* class. *VerifyUser* utilizes the *System.getProperty()* method, with the argument *user.name*. *VerifyUser* compares the result from *System.getProperty("user.name")* with input to the program and returns the value depending on the result the comparison. Appendix B contains the entire source code for TargetUser.

The ability to use different programming languages to develop modules for MAST allows each language to have a different set of libraries that can be utilized for their specific functionality. The remaining three modules developed for this scenario were written in Python 3.6 using its standard library as well as imported libraries to support specific functionalities. One of the areas for future work that was identified by Aybar (2017) was the need for “looping constructs, complex conditionals, or variable assignments” within the MAST framework (65). When a command is received by the client, it executes and returns the result. If the module were checking for a specific condition or waiting for an event, due to network latency or hosts that are slow to open a program, the module may

miss the event and return a false-negative result. To overcome this, we built looping constructs into the remaining modules to account for latency and other delays that may occur.

The AppRunning module takes as input an application name with no spaces and returns whether the application is a running process on the host. This is used in the scenario to check whether the user has clicked on the link in the phishing email to open a browser and initiate a connection to the attacker's web server. AppRunning imports two libraries, the `sys` and `psutil` libraries. The `sys` library provides access to the program's arguments and `psutil` is an extensive and convenient library that is used for "retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python" (pypi.org n.d.). AppRunning is comprised of four functions; *checkinput()*, *checkrunning()*, *notfound()*, and *filefound()*. AppRunning also includes a dictionary of common application names and the associated name of their executables to assist mapping the arguments to processes running. The *checkinput()* function validates the arguments to the program and provides feedback if the argument is not valid, if the input is valid it attempts to map the argument to its executable name and calls the *checkRunning()* function.

The *checkRunning()* function utilizes the *psutil.process_iter()* function from the `psutil` library to generate a list of the running processes. Using this, we generate our own list of processes and their process identification numbers. We then iterate through the list to check for the target program; if found, we exit with the return value of 1. If the process is not found, *checkRunning()* will loop through the process list 5000 times, which depending on the number of processes running takes between 180 and 300 seconds. If the process is eventually not matched, it calls the *notfound()* function and exits with a value of 0. Appendix C contains the entire source code for AppRunning.

The InetConnection module checks for an Internet connection between the host and the IP address passed into program. A connection is required between the target and the attacker's server in order to deliver the payload. The InetConnection module takes one argument, an IP address in dot-decimal notation and returns 1 if there is a connection, 0 if there is no connection or 456 if there is an error. InetConnection also utilizes the `psutil` and `sys` libraries. The program is comprised of four functions; *checkInput()*, *checkloop()*,

getConnections() and *verifyConnections()*. *CheckInput()* verifies the input to the program and provides feedback if the input is not valid, if the input is valid it sets the target IP address variable with the programs input. The *checkloop()* function is the overall control loop for the program and loops 2000 times or until the target IP is detected as connected. *Checkloop()* calls *getConnections()* which utilizes the *psutil.net_connections()* function to generate a list of all network connections as tuples that are active on the host. We then parse the list of connections and build a dictionary of remote IP addresses to which the host is connected. *Checkloop()* then calls *verifyConnections()* to execute a dictionary lookup of target IP address and returns 1 if the connection is detected, or 0 if not. Similar to *AppRunning*, the loop accounts for latency within network and allows the scenario to execute even if events are not perfectly timed. Appendix D contains the entire source code for *InetConnection*.

The previous four modules that we described are examples of condition modules that check for a specific condition or event to occur on the target. The last module that was developed for this scenario is an example of a *SimWare* module that simulates the effects and behaviors of malware, but does not cause any actual damage to system. The *LockOut* module serves as our payload, and is designed to disrupt the C2 network by locking the users out of their computers and prevent them from viewing any of the security feeds they are maybe monitoring. This will enable the ground forces to maintain the element of surprise as they begin their raid on the base.

The inspiration for *LockOut* comes from recent real-world ransomware attacks that prevent users from accessing data on their devices by encrypting the data with a secret key. The attackers provide the decryption key only after a ransom is paid by the victim. *LockOut* is similar in that it implements a full-screen, top-level Graphical User Interface (GUI) that cannot be closed and no other applications can be accessed or viewed without entering the correct password. *LockOut* is written in Python and utilizes the *tkinter*, *sys*, *time*, and *hashlib* libraries. It takes one argument - the password that must be used to unlock the screen and close the program, stored as an MD5 hash digest. *LockOut* is comprised of three functions; *init()*, *getPassword()* and *quit()*. The *init()* function builds the GUI with the *tkinter* library, and sets the GUI with the “full-screen,” “topmost,” and “overridedirect”

configurations, which force the GUI to remain on top of all other applications and removes the traditional close, minimize and exit buttons from the GUI. It also implements an entry field for the user to enter the password and provides feedback when the wrong password has been entered.

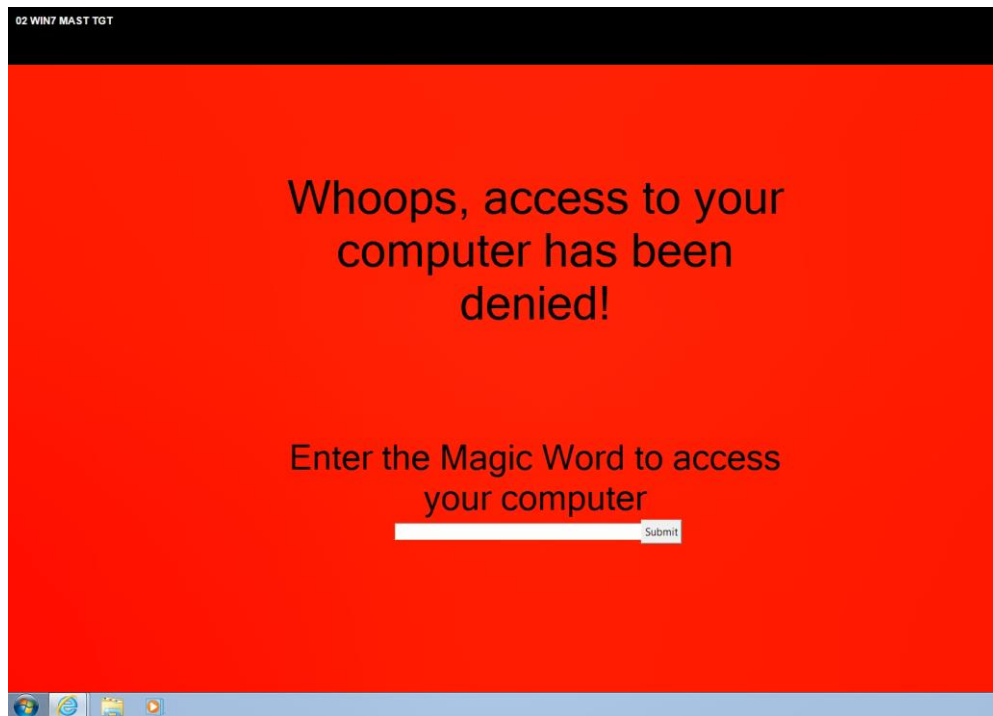


Figure 13. LockOut SimWare Module

The configuration of the GUI, as seen in Figure 13, forces the GUI to appear over all other applications, the start menu can be accessed but any other window including the command prompt and task manager appear behind the LockOut and cannot be accessed. The *getPassword()* function takes the value that the user enters into the entry box, hashes it, and compares its digest to the digest of the password passed into the program. If the digests match, the *getPassword()* function calls the *quit()* function to destroy the GUI and exit the program. Appendix E contains the entire source code for LockOut.

AppRunning, InetConnection, and LockOut are all written in Python, which would require each MAST client to also have Python and the imported libraries installed to run

the programs. To overcome this and increase portability of the modules across platforms, we use the PyInstaller application to “freeze the program into a standalone executable” which includes all libraries and imports necessary to run the program from the MAST client, running on any Windows, Linux, or OSX platform. (Pyinstaller 2018). In addition to the modules that were developed for this scenario, our attack model also requires the WriteEicar, DriveByDownload and LogicBomb, which are three modules that are currently available in MAST previously described in Chapter III. Combining our new modules with those currently available in MAST, we have all modules necessary to complete our attack model, which can now be tested with MAST. The following sections detail the test environment, the test network and scenario script that simulate the execution of the attack.

B. TESTING

1. Cyber Battle Lab

The NPS Cyber Battle Lab (CYBL) is a Type I hypervisor that was previously used by Aybar (2017) as an environment for testing MAST. The CYBL provides a support structure for building complete virtual networks that can be reconfigured as each scenario dictates, and it has the computing resources to support large-scale network virtualization. The CYBL also has a large selection of virtual machine templates including servers, routers, and a range of operating systems to support replicating almost any network. The lab is remotely accessed through the VMware Horizon Client desktop application or through a web browser interface, which enables development and implementation from remote locations.

In order to test our attack model, we replicate the target network in the CYBL on two subnets to build the C2 and DMZ networks with all identified hosts and servers. We establish all services that are running on the target network including mail, DNS, active directory, routing, NAT and access control lists (ACL) to model the network conditions and connections. We also isolate this network from the Internet to prevent any unintended updates or changes that may affect our test environment.

2. Target Network

As shown in Figure 14, the test network is divided into two subnets: Aybar Team 1 - a virtual network that represents the C2 Network, and Aybar Team 2 - a virtual network that represents the DMZ. Each host on Aybar Team 1 is a Windows 7 SP1 virtual machine with the following configuration:

- 32 GB Disk
- 2048 MB RAM
- Eclipse IDE for Java Developers: Neon.3
- Java SE Development Kit 8
- Thunderbird Mail Client 52.7.0
- ExQuilla for Microsoft Exchange Add-On
- Internet Explorer version 11.09
- AVG Anti-Virus version 17.3443
- MAST Client with all available modules

All hosts on Aybar Team 1 can also access the MASTFiles FTP server located within the MAST Admin and Scenario Control section of the subnet. The MASTFiles server hosts the MAST source code, new modules and applications that a MAST Client may need. Since we isolate the test network from the Internet, we use the MASTFiles server as a repository for MAST Clients to pull updated code without having to access the Internet. The MASTFiles server also hosts the malicious web server for the scenario. MAST Admin & Scenario Control also hosts WIN7-TGT-01, which is a Windows 7 virtual machine that serves as the SGS and SES for the MAST scenarios. It issues commands to the clients to execute according to the scenario script. Connecting the two subnets is an Ubuntu 14.04 LTS virtual machine with routing enabled that implements an ACL that blocks and logs externally generated traffic from entering the Aybar Team 1 subnet.

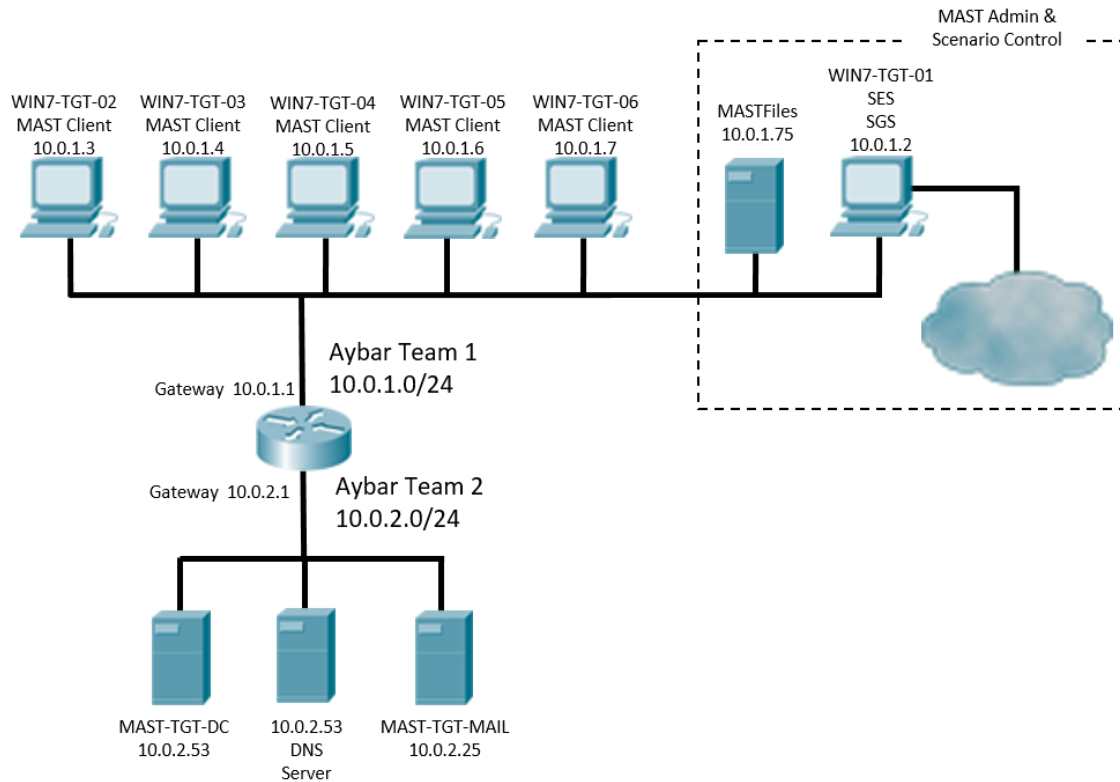


Figure 14. CYBL Test Network

Aybar Team 2 hosts the DMZ, which contains two Windows Server 2012 virtual machines. The MAST-TGT-DC has two roles, the domain controller and DNS server for the test network, as shown in Figure 14, but both reside within the same virtual machine and share the same IP. Both are fully functional, providing active directory and domain name resolution for the mast.cyber domain. The MAST-TGT-MAIL server provides both Simple Mail Transfer Protocol (SMTP) and Internet Message Access Protocol (IMAP) services with Exchange Server 2013 to the clients in the mast.cyber domain. Clients access their accounts with the Thunderbird mail client configured with the ExQuilla Exchange add-on.

3. Attack Scenario Scripting

a. Scenario

The script starts with the [Scenario] header that identifies the name of the scenario and the minimum number of clients. The scenario writer determines the minimum number of clients; it must be at least one. In scenarios that require more than one client, the SES

checks that the minimum number of clients is connected before the scenario can start. For our scenario, as long as one client is connected, the scenario will execute.

b. Module List

The [ModuleList] identifies all modules the clients are required to execute during the scenario. When a client connects, it passes to the SES the modules that are installed, and the SES determines which clients can be used in a scenario. As described in our attack model, we use the TargetSpecificOS, AppRunning, DriveByDownload, LockOut, InetConnection, WriteEicar, and LogicBomb modules.

c. Group List

The [GroupList] allows the scenario writer to group clients in order to send specific clients a command. The client can also change groups when an event occurs or on command from the SES. All clients start out in Group 1, and a client transitions to Group 2 if the user clicks the link in the phishing email and opens the browser. The client moves to Group 3 if the user accepts the file to download.

d. Infected

The [Infected] section identifies a group that are represented as infected in the MAST GUI. In our scenario, the only infected group is Group 3, after a user accepts to download the malicious file, the SES moves that client to Group 3 and the client is identified as infected on the MAST GUI.

e. Command List

The [CommandList] represents the numbered list of commands that the SES sends to each client. The commands are referenced to by their number in the command list and passed in the format <module arg1 arg2>. The first command in the [Command List] is interpreted as execute the TargetSpecificOS module with Windows 7 as its arguments.

f. Events

The [Events] section brings together the commands, the modules, and groups in order to simulate the execution of the attack path. We explain below each event and relate its execution to the context of the scenario.

1. T 0 SGC 1 1 is the first event in any MAST Scenario. It starts the scenario and at Time 0 sends all clients in Group 1, command 1 in the [CommandList] which is TargetSpecificOS Windows 7. This checks that client machine is running Windows 7, which our payload is built for.
2. RC 1 TargetSpecificOS 1 SCC checks the return code (RC) from TargetSpecificOS that was sent to Group 1. All clients that return 1 have Windows 7 running are then sent command 2, AppRunning Internet Explorer.
3. RC 1 AppRunning 1 SCC 3 checks the RC from AppRunning that was sent to all clients with Windows 7 installed. All clients that return 1, meaning Internet Explorer is running are sent command 3, InetConnection 10.0.1.75.
4. RC 1 AppRunning 1 CG 2 moves all clients that responded that Internet Explorer was running to Group 2. This allows us to direct commands only to the group that opened the link in their browser.
5. RC 2 InetConnection 1 SCC 4 checks all clients in Group 2 for an Internet connection to the malicious server at IP 10.0.1.75. If the connection exists, those clients are sent command 4, DriveByDownload.
6. RC 2 DriveByDownload 1 SCC 5 checks the RC from all clients in Group 2. If those clients accept the download, they return 1 and are sent command 5, WriteEicar which downloads a file to the client.
7. RC 2 DriveByDownload 1 CG 3 moves all clients that accept the download to Group 3.

8. RC 3 WriteEicar 2 SCC 6 checks that the file was downloaded to the client. If the file was downloaded, the client returns 1 and is sent command 6, LogicBomb.
9. RC 3 LogicBomb 1 SCC 7; the LogicBomb returns 1 when the argument time is reached. This represents the time we want our payload to execute and achieve the desired effects on the target. When the clients in Group 3 return 1 from the LogicBomb, they are sent command 7 to execute the payload LockOut.
10. RC 3 LogicBomb 1 SGC 1 5 is used to simulate propagation of our malware across the entire C2 network. Once one user accepts the download and the designated time has been reached, the malware spreads and is written to clients in Group 1.
11. RC 3 LogicBomb 1 SGC 2 5 is the same as event 10, but the malware is propagating to those clients in Group 2.
12. RC 1 WriteEicar 2 SGC 1 7 checks that malware was written to clients in Group 1 and sends them command 7 to execute the LockOut.
13. RC 1 WriteEicar SGC 2 7 checks the malware was written to clients in Group 2 and sends them command 7 to execute the LockOut.
14. RC 1 WriteEicar 2 CG 3 moves those clients in Group 1 to the infected Group.
15. RC 2 WriteEicar 2 CG 3 moves those clients in Group 2 to the infected Group.

```

[Scenario]
    Name=DegradeC2Net
    MinClients=1

[ModuleList]
    1=AppRunning
    2=DriveByDownload
    3=LockOut
    4=InetConnection
    5=WriteEicar
    6=LogicBomb
    7=TargetSpecificOS

[GroupList]
    1=100%
    2=0
    3=0

[Infected]
    1=3

[CommandList]
    1=TargetSpecificOS Windows 7
    2=AppRunning InternetExplorer
    3=InetConnection 10.0.1.75
    4=DriveByDownload
    5=WriteEicar
    6=LogicBomb 2018-05-12 20:37:00
    7=LockOut mast

[Events]
    1=T 0 SGC 1 1
    2=RC 1 TargetSpecificOS 1 SCC 2
    3=RC 1 AppRunning 1 SCC 3
    4=RC 1 AppRunning 1 CG 2
    5=RC 2 InetConnection 1 SCC 4
    6=RC 2 DriveByDownload 1 SCC 5
    7=RC 2 DriveByDownload 1 CG 3
    8=RC 3 WriteEicar 2 SCC 6
    9=RC 3 LogicBomb 1 SCC 7
    10=RC 3 LogicBomb 1 SGC 1 5
    11=RC 3 LogicBomb 1 SGC 2 5
    12=RC 1 WriteEicar 2 SGC 1 7
    13=RC 2 WriteEicar 2 SGC 2 7
    14=RC 1 WriteEicar 2 CG 3
    15=RC 2 WriteEicar 2 CG 3

```

Figure 15. Test Scenario Script

4. Attack Simulation

When SES executes the scenario, we can observe the flow of events in our attack path, checking each condition and observing the effects on each target. Beginning with the attack in the initial attack state, Figure 16 shows the MAST GUI, controlled by SES, prior to the start of the scenario. The attack transitions when the phishing email, shown in Figure 17, is received by a user on the C2 network. When a user opens the email, he is presented with a seemingly official email, with a deadline to complete a task and a reference to a staff meeting that never occurred. The phishing email uses common social engineering techniques that unknowingly pressure and deceive the user into clicking the link and transition the attack. When the user clicks the link in the phishing email to open a web browser and connect to the malicious server, we can see the result on the user's desktop in Figure 18.

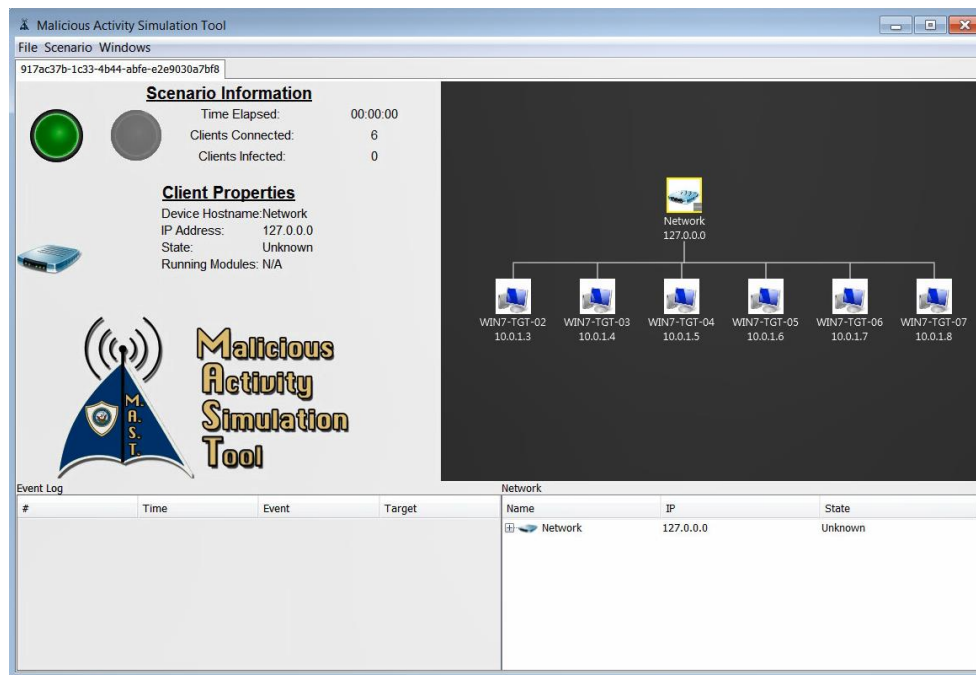


Figure 16. MAST GUI Prior to the Scenario Start

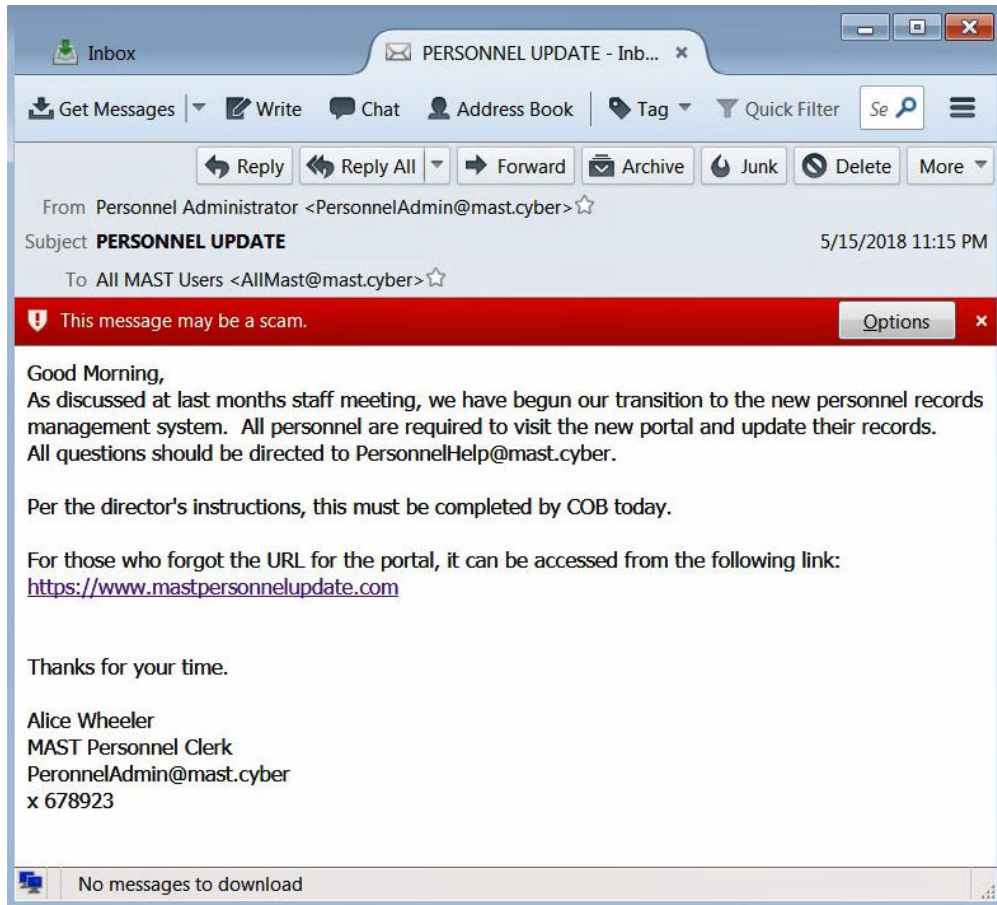


Figure 17. Phishing Email Sent to Target Network Users



Welcome to the MAST Personnel Portal

Please gather all of your personnel information which includes:

- Your Employee ID Number
- The Name of your first pet
- Your mother's maiden name
- The street you grew up on
- Your last highschool mascot



When prompted please download the requested file to start the update



Figure 18. Attacker's Web Server and Download Warning

When a user is presented with the website that relates to the email he received, it is not uncommon for an untrained user to accept the download that delivers payloads to their system. Using the LogicBomb, we set the time for the payload to execute to coincide with the commencement of the ground attack. When that time is reached, the malware executes on the system as seen earlier in Figure 13, and then propagates across the C2 network disrupting the adversary's ability to detect our ground attack. The result of our attack is shown in Figure 19, with all clients infected and denied access to their devices with the LockOut SimWare. Figure 20 demonstrates the attack on a single host, with the event log detailing each module being run in sequence with the attack finishing in the end state and the user locked out.

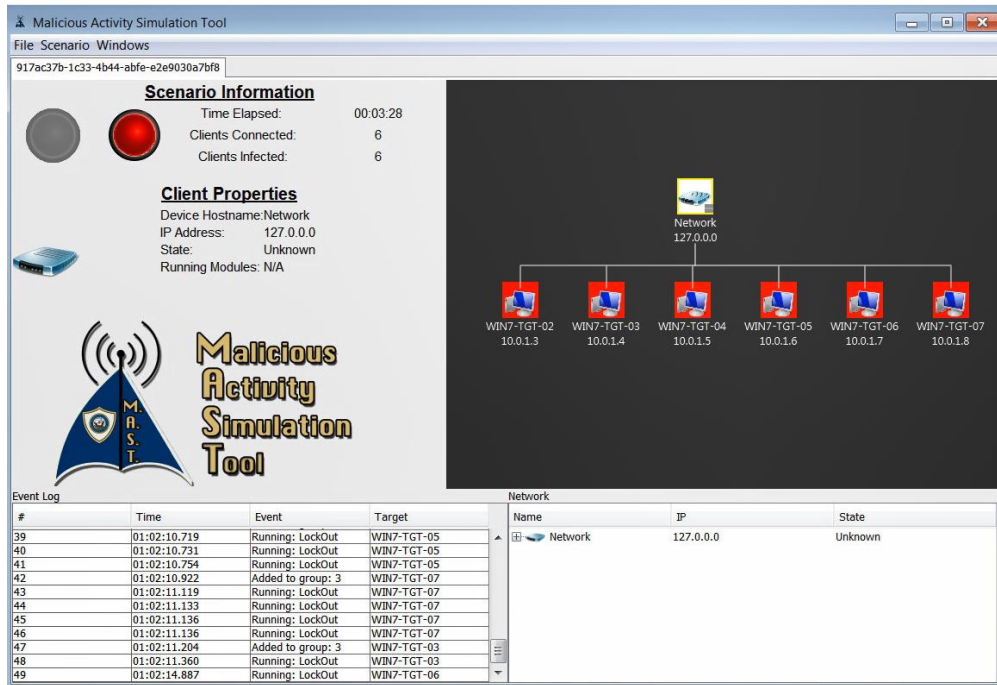


Figure 19. MAST GUI with All Targets Infected

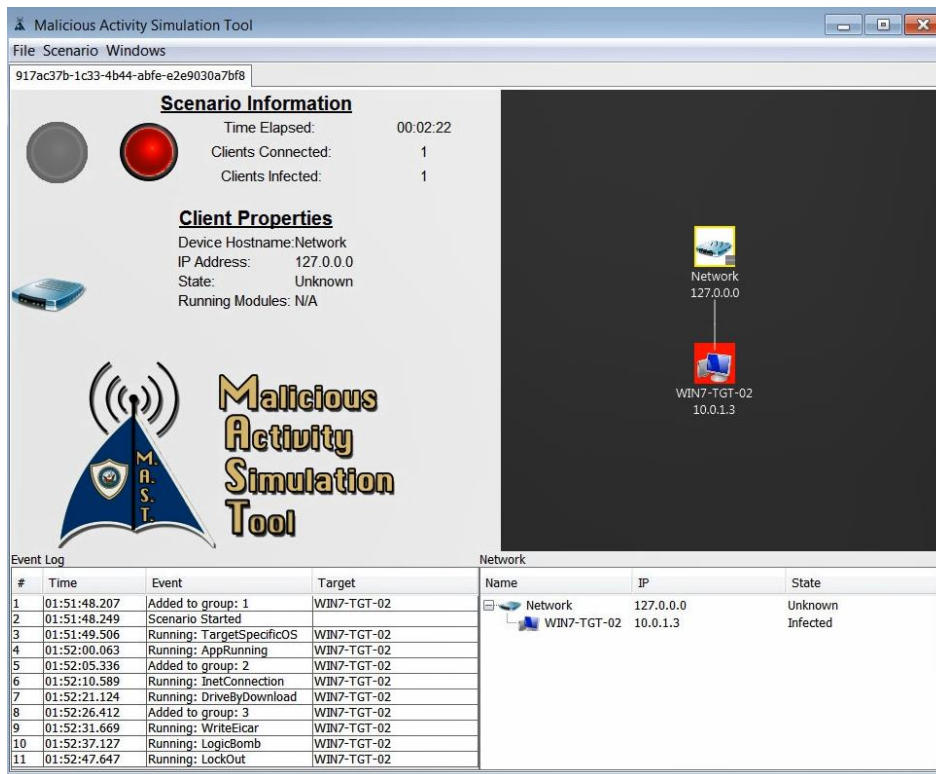


Figure 20. Complete Event Log with a Single Host Infected

C. SUMMARY

Using an example scenario, this chapter describes how we identify vulnerabilities and generate an attack graph for targeting a system. We generated an attack model for the chosen attack path, identifying the states and transitions necessary to reach the desired end state. We detailed the development of the modules necessary to simulate our attack model in the MAST framework. We described our test environment, our test network and the scenario script used to simulate our attack path. We then execute our scenario script on our test network and observe the effects on the targeted systems while also viewing the scenario events executing in sequence in the MAST GUI.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND FUTURE WORK

A. SUMMARY

The primary goal of this thesis was to leverage and build on previous research by extending the MAST framework and implementation as a platform for simulating cyberspace attack scenarios, and to enable future advances in its development. We developed a methodology for modeling a cyberspace attack by breaking it down into its basic elements, and then representing the attack as a finite state machine of atomic events. We also defined the conditions that must be met for an attack to transition through the state machine. The condition categories were defined to focus, but not limit, development of future cyberspace attack modules.

In order to validate our model, we designed an example scenario to demonstrate the development of a cyberspace attack from its high-level objectives by breaking it down into its attack model represented as a finite state machine. We used the model to identify modules that would need to be developed to simulate the scenario within MAST. From this demonstration, five new SimWare modules were added to the MAST framework: TargetSpecificOS, TargetUser, AppRunning, InetConnection, and LockOut. The LockOut module was developed to fulfill one of the original intentions of MAST, which was to simulate the effects of malware without actually causing damage to a system.

B. CONCLUSIONS

The objectives and goals of this thesis were successfully accomplished. The model we developed, along with five new SimWare modules, can be used in future MAST scenarios to build larger and more complex cyberspace attacks from smaller, more manageable elements. The development of a more robust and realistic target network will also enable future extensions to MAST.

The primary question that we sought to answer with this thesis was: how can offensive cyberspace attacks be modeled to effectively simulate and rehearse their execution against a target network? The development of an attack model, represented as a finite state machine, to model cyberspace attacks demonstrated that we can model large

and complex attack scenarios with a sequence of smaller, less complex functions or modules. We can then use MAST to simulate those sequences or attack paths, and either observe the effects that we want to achieve or identify gaps in our attack that were not considered. We demonstrated this capability with an example implementation scenario that broke a complex attack into its attack model, represented as a finite state machine, and then successfully simulated that machine with MAST to observe its effects.

The secondary questions we sought to answer were: what techniques are used to simulate the effects of a denial-of-service attack on an adversary network; and, what techniques are used to simulate the sequencing of offensive cyberspace operations? In order to demonstrate the effects of a cyberspace attack we developed the LockOut SimWare module, which simulates a denial-of-service attack by denying a user access to their system. For the MAST simulation, we identified the behavior that must be simulated, but not the actual method, as the method is most often malicious and can cause damage to the system. The behavior, or the effect of the behavior, can often be achieved by other means, such as our GUI implementation to deny access as opposed to full disk encryption to deny access. The sequencing of a cyberspace operation comes down to identifying what events need to happen, and in what order, for the attack to be successful. The attack model that we developed from an identified attack path is an ordered sequence of events that must occur in order for the attack to be successful. We demonstrated with our implementation that we can simulate the correct sequence of operations for a successful cyberspace attack.

C. RECOMMENDATIONS FOR FUTURE WORK

The evolution of MAST has shown how flexible it is for taking on different roles in cyber operations. Its use as a platform for developing cyberspace attacks is relatively new, as compared to its original design as a training platform for system administrators. Despite the advancements made by this thesis, there are still numerous ways that MAST can be improved upon and extended.

1. Graphical User Interface for Scenario Scripting

As the number of modules will continue to grow and scenarios become more complex, it would be beneficial if the scenario script could be generated through a GUI

input rather than the current method of a formatted script text file. This would decrease user-entry mistakes, save time searching for errors, and make scenario development clearer to the scenario writer, e.g., by providing information about specific commands.

2. Client Program Improvement

Process Builder is an effective means for launching sub-processes within Eclipse, as long as the sub-processes do not have any dependencies outside of the executable file, such as images, data sets, etc. An improvement to the MAST Client would be to incorporate a working directory where dependencies can be stored so that modules can access them in real-time. Additional logging capability would also benefit scenarios when modules are not working correctly.

3. Additional Modules

SimWare modules are the basis of MAST, and allow a programmer to get creative and figure out different ways to effectively simulate a behavior without actually damaging a system. This is an area where MAST needs to be extended further.

4. MAST GUI

The MAST GIU can be improved to include visualization of the attack graphs that are generated for each scenario. The current event log can become difficult to understand at scenarios become more complex. Interaction with the targets within the GUI could provide additional information about the specific targets or possibly issue additional commands to a specific target.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. TARGET SPECIFIC OS SOURCE CODE

```
package TargetSpecificOS;

public class TargetSpecificOS {

    public static void main(String[] args) {

        int OpSysArg = 0;
        int OSVerArg =1;
        int errorCode = 456;
        String targetOS="";
        String OSVersion="";
        int returnCode = errorCode;

        if (args.length == 2){

            targetOS = args[OpSysArg].toLowerCase();
            OSVersion = args[OSVerArg];
            System.out.printf("A Machine with OS: %s %s will be
targeted\n",targetOS,OSVersion);
        }
        else {
            System.err.println("Invalid Input. Must be in the format:
OperatingSystem Version");
            System.err.println("Example: Windows XP");
            System.err.println("Exiting with Error Code");
            returnCode = errorCode;
            System.exit(returnCode);
        }

        VerifyOS osCheck = new VerifyOS();
        returnCode = osCheck.VerifyOSCheck(targetOS,OSVersion);
        System.out.print(returnCode);
        System.exit(returnCode);
    }
}

package TargetSpecificOS;

public class VerifyOS {

    public VerifyOS(){
    }

    public int VerifyOSCheck(String targetOS, String OSversion){

        int OSComp = -1;
        int verComp =-1;
        int errorCode = 456;
        int returnCode = errorCode;
```

```

    int successCode = 1;

    String OSfull = System.getProperty("os.name");

    String target= String.join(" ", targetOS,OSversion);
    target = target.toLowerCase();

    String[] OSstring = OSfull.split(" ");
    String OS = OSstring[0].toString();
    OS = OS.toLowerCase();
    String OSver = OSstring[1].toString();
    OSver = OSver.toLowerCase();

    OSComp = targetOS.compareTo(OS);
    verComp = OSversion.compareTo(OSver);

    if (OSComp ==0 && verComp ==0){
        System.out.print("Success. Match for Targeted OS and
        Version\n");
        returnCode = successCode;
    }

    if (OSComp ==0 && verComp !=0){
        System.out.print("Failure. Match for Targeted OS, version
        was not found\n");
        returnCode = errorCode;
    }

    if (OSComp !=0 && verComp ==0){
        System.out.print("Failure. No match for Targeted OS type,
        version was matched\n");
        returnCode = errorCode;
    }

    if (OSComp !=0 && verComp !=0){
        System.out.print("Failure. No match for Targeted OS or
        version\n");
        returnCode = errorCode;
    }

    return returnCode;
}
}

```

APPENDIX B. TARGET USER SOURCE CODE

```
public class TargetUser {

    public static void main(String[] args) {

        int firstNameArg = 0;
        int lastNameArg = 1;
        int errorCode = 456;
        String targetUser="";
        int returnCode = errorCode;

        if (args.length == 2){

            targetUser = String.join(" ",
args[firstNameArg],args[lastNameArg]);
            targetUser = targetUser.toLowerCase();
            System.out.printf("%s\n",targetUser);
            System.out.printf("A Machine with user %s logged in
will be targeted\n",targetUser);
        }

        if (args.length == 1){
            targetUser=args[firstNameArg];
            targetUser = targetUser.toLowerCase();
            System.out.printf("%s\n",targetUser);
            System.out.printf("A Machine with user %s logged in
will be targeted\n",targetUser);
        }

        if (args.length == 0) {

            System.err.println("Invalid Input. Must be in the
format: Username");
            System.err.println("Example: Paul Allen");
            System.err.println("Exiting with Error Code");
            returnCode = errorCode;
            System.exit(returnCode);
        }

        VerifyUser userCheck = new VerifyUser();
        returnCode = userCheck.VerifyUserCheck(targetUser);
        System.out.print(returnCode);
        System.exit(returnCode);
    }
}
```

```

public class VerifyUser {

    public VerifyUser(){

    }

    public int VerifyUserCheck(String targetUser){
        int successCode = 1;
        int errorCode = 456;
        int userComp = -1;
        int returnCode = errorCode;

        String user = System.getProperty("user.name");

        user=user.toLowerCase();

        userComp = user.compareTo(targetUser);

        if (userComp == 0){
            System.out.print("Success. Targeted User Identified\n");
            returnCode=successCode;
            return returnCode;
        }
        else {
            System.out.print("Failure. Targeted User was not
identified\n");
            returnCode=errorCode;
        }
        return returnCode;
    }

}

```

APPENDIX C. APP RUNNING SOURCE CODE

```
import psutil
import sys
import time

ProcNames={'internetexplorer':'iexplore.exe', 'googlechrome':'chrome.exe',
'chrome':'chrome.exe','excel':'excel.exe'}

errorCode = 456
notFoundCode=0
successCode= 1
returnCode=0
appName=''

def checkinput():

    if len(sys.argv) != 2:
        print("Invalid Input. Must be in the format: ProgramName\n")
        print("Example: InternetExplorer")
        print("Exiting with Error Code")
        returnCode=errorCode
        sys.exit(returnCode)
        print(returnCode)

    else:
        appName=sys.argv[1].lower()

        try:
            appName=ProcNames[appName]
            checkRunning(appName);

        except KeyError:
            print("AppName not defined, will try anyway but the
            application may not be found even if its running")
            checkRunning(appName)

def checkRunning(App):
    numofloops=5000

    while numofloops !=0:

        for processes in psutil.process_iter(attrs=['pid','name']):
            if App in processes.info['name']:
                filefound()
                numofloops-=1
        notfound()

def notfound():

    returnCode=notFoundCode
    print(returnCode)
    sys.exit(returnCode)
```

```
def filefound():  
    returnCode=successCode  
    print(returnCode)  
    sys.exit(returnCode)  
  
if __name__=="__main__":  
    checkinput()
```

APPENDIX D. INET CONNECTION SOURCE CODE

```
import psutil
import sys

targetIP=[]
connectionList=[]
ipaddrstable={}
successCode=1
failureCode=0
returnCode=0
errorCode=456

def checkInput():
    if len(sys.argv) != 2:
        print("Invalid Input. Must be in the format: XXX.XXX.XXX.XXX\n")
        print("Example: 123.456.789.101")
        print("Exiting with Error Code")
        returnCode=errorCode
        sys.exit(returnCode)
        print(returnCode)
    else:
        targetIP.append(sys.argv[1])

def checkloop():
    numofloops=2000

    while numofloops != 0:

        getConnections()
        ret = verifyConnection()

        if ret == successCode:

            returnCode=successCode
            print(returnCode)
            sys.exit(returnCode)

        else:
            numofloops-=1
            connectionList.clear()

    returnCode=failureCode
    print(returnCode)
    sys.exit(returnCode)

def getConnections():

    for connections in psutil.net_connections():
        connectionList.append(connections)

    for data in connectionList:
```

```

        remoteAddress=data[4]
        if remoteAddress:
            ipaddress=remoteAddress[0]
            try:
                ipaddrstable[ipaddress] += 1

            except KeyError:
                ipaddrstable[ipaddress] =1

def verifyConnection():

    try:
        ipaddrstable[targetIP[0]] +=1

        return 1

    except KeyError:

        return 0

if __name__=="__main__":
    checkInput()
    checkloop()

```

APPENDIX E. LOCK OUT SOURCE CODE

```
import tkinter
import sys
import time
import hashlib

base=tkinter.Tk()

passcode=hashlib.md5(sys.argv[1].encode('utf-8')).hexdigest()

class Lockout:

    def __init__(self, master=None):
        self.base=master
        self.base.title("LockOut")
        self.base.attributes('-fullscreen',True)
        self.base.attributes('-topmost', True)
        self.base.overrideredirect(True)
        self.base.configure(background="red")
        self.frame=tkinter.Frame(base, background="red")
        self.frame.grid()
        self.label1text='Whoops, access to your computer has been denied!'
        self.label1=tkinter.Label(self.frame,text=self.label1text,
        font=("Microsoft Sans Serif",40), fg = "black", bg = "red", justify=
        "center", wraplength='800', padx=400, pady=150)

        self.label1.grid(row=0,column = 1, columnspan=4)
        self.label2=tkinter.Label(self.frame,text = "Enter the Magic Word to
        access your computer", font=("Microsoft Sans Serif",30), fg =
        "black", bg = "red", justify= "center",wraplength='800')

        self.label2.grid(row = 1, column=1, columnspan=4)
        self.entry=tkinter.Entry(self.frame,show="*", width=50)
        self.entry.grid(row =5 ,column=1, columnspan=4)
        self.ebutton=tkinter.Button(self.frame, text="Submit", command=
        lambda arg = None: self.__getPassword__())

        self.ebutton.grid(row = 5, column=3)

    def __getPassword__(self):

        userentry=self.entry.get()
        if hashlib.md5(userentry.encode('utf-8')).hexdigest() ==
        passcode:
            self.__quit__()

        else:
            self.label1.config(text = "Ah, Ah, Ah..You didn't say
            the Magic Word!")
            self.label1.update()
            time.sleep(5)
            self.label1.config(text = 'Whoops, access to your
            computer has been denied!')
```

```
def __quit__(self):  
    self.base.destroy()  
  
if __name__=="__main__":  
    run=Lockout(base)  
    base.mainloop()
```

LIST OF REFERENCES

- Ablon, Lillian, and Andy Bogart. 2017. *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits*. RR-1751-RC. Santa Monica, CA: RAND. https://www.rand.org/pubs/research_reports/RR1751.html.
- Aybar, Luis E. 2017. "Developing Simulated Cyber Attack Scenarios Against Virtualized Adversary Networks." Master's thesis, Naval Postgraduate School. <https://calhoun.nps.edu/handle/10945/52999>.
- Belli, Gregory F. 2016 "Extensible Simware Architecture for Flexible Training Scenarios." Master's thesis, Naval Postgraduate School.
- Collier, Anthony R. 2016. "Automated Network Mapping and Topology Verification." Master's thesis, Naval Postgraduate School. <https://calhoun.nps.edu/handle/10945/49438>.
- Cvedetails.com. 2015. "Windows OLE remote code execution vulnerability - CVE-2014-4114 (MS14-060)." October 8, 2015. <https://www.cvedetails.com/cve/CVE-2014-4114/>.
- Denning, Peter J., and Dorothy E. Denning. 2010. "Discussing Cyber Attack." Comm. of the ACM 53, no. 9 (September): <http://hdl.handle.net/10945/37166>.
- Engelbreton, Patrick. 2013. *The Basics of Hacking and Penetration Testing*. Waltham, MA: Syngress.
- Falliere, Nicholas, Liam O Murchu, and Eric Chen. 2011. *W32.Stuxnet Dossier*. Cupertino CA: Symantec Corporation. https://www.symantec.com/content/en/us/enterprise/.../w32_stuxnet_dossier.pdf.
- Jajodia, Sushil and Steven Noel. 2010. *Advanced Cyber Attack Modeling, Analysis and Visualization*. Report No. AFRL-RI-RS-TR-2010-078. handle.dtic.mil/100.2/ADA516716&embedded=true.
- Joint Chiefs of Staff. 2010. *Department of Defense Dictionary of Military and Associated Terms*. JP 1-02. Washington, DC: Joint Chiefs of Staff. https://fas.org/irp/doddir/dod/jp1_02.pdf.
- Joint Chiefs of Staff. 2013a. *Cyberspace Operations*. JP3-12R. Washington, DC: Joint Chiefs of Staff. www.jcs.mil/Portals/36/Documents/Doctrine/pubs/jp3_12R.pdf.
- Joint Chiefs of Staff. 2013b. *Joint Targeting*. JP3-60. Washington, DC: Joint Chiefs of Staff. <http://www.jcs.mil/Doctrine/Joint-Doctrine-Pubs/3-0-Operations-Series/>.

- Jha, S., O. Sheyner and J. Wing. 2002. “Two Formal Analyses of Attack Graphs” In *Proceedings 15th IEEE Computer Security Foundations Workshop*: 49–63. DOI: 10.1109/CSFW.2002.1021806
- Kennedy, David, Jim O’Gorman, Devon Kearns, and Mati Aharoni. 2011. *Metasploit The Penetration Tester’s Guide*. San Francisco, CA: No Starch Press.
- Lowney, Erik S. 2015. “Network Communications Protocol for the Malicious Activity Simulation Tool (MAST).” Master’s thesis, Naval Postgraduate School.
- Microsoft. 2017. “Microsoft Security Bulletin MS14-060 – Important.” October 11, 2017. <https://docs.microsoft.com/en-us/security-updates/securitybulletins/2014/ms14-060#vulnerability-in-windows-ole-could-allow-remote-code-execution-3000869>.
- Neff, Justin M. 2012. “Verification and Validation of the Malicious Activity Simulation Tool (MAST) for Network Administrator Training and Evaluation.” Master’s thesis, Naval Postgraduate School. <https://calhoun.nps.edu/handle/10945/6842>.
- Nieves, Michael, Kelley Dempsey, and Victoria Pillitteri. 2017. *An Introduction to Information Security*. National Institute of Standards and Technology. 800–12 Rev. 1. Gaithersburg, MA: NIST. <https://doi.org/10.6028/NIST.SP.800-12r1>.
- PyInstaller. 2018. “PyInstaller – Welcome to the PyInstaller Official Website.” February 25, 2018. <https://www.pyinstaller.org/#>.
- Pypi. n.d. “psutil 5.4.5.” Accessed April 28, 2018. <https://pypi.org/project/psutil/>.
- Taff, William, and Paul. Salevski 2011. “Malware Mimics for Network Security Assessment.” Master’s thesis, Naval Postgraduate School, Monterey, CA, <https://calhoun.nps.edu/handle/10945/5749>.

INITIAL DISTRIBUTION LIST

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