Matthew Collins
Philippe Lessard
Jacob Tutlis
CS 4404 - B 19
Mission 3 - Novel IDS IDeaS

INTRODUCTION

For this mission, we designed a command and control system that sent data through covert channels using NTP packets. NTP is the Network Time Protocol that is used to synchronize the clocks between computers. NTP packets have a reference ID field that is used to identify the IP of the source server, however, we used this field to transmit our commands to appear as an IP address in this field. The infected host reads the hidden instructions in the NTP packets and launches an attack on another machine, with the IP address of that machine being sent in the same NTP packets.

For our defense, we were able to analyze the time between NTP packets and determine if those packets met the distinct pattern in which NTP packets are sent. We were also able to analyze the reference ID field and determine if the IP address was a known valid NTP server.

RECONNAISSANCE

When an attacker gains access to a computer network by compromising a host on the network, they are potentially able to do significant damage to the network from within. Alternatively, the resources and higher bandwidth levels of enterprise networks might allow an attacker to create a powerful botnet to attack computers outside the network. In order to maintain control of the compromised computer for long, the attacker must remain undetected to prevent the compromised computer from being wiped. However, in most cases, the attacker will still want to coordinate attacks and other malicious actions on the device through a command and control system. On the defensive side, a network administrator would want to be able to detect these command and control messages that are being sent over the network, and there are various techniques that can be used to detect malicious activity. Network Intrusion Detection Systems (NIDS) can come with various capabilities but usually only make use of one technique between Deep Packet Inspection (DPI), network flow detection, and SDN-based monitoring.

Deep Packet Inspection

Deep Packet Inspection is a technique where packets routed through a network node are sniffed, allowing the entire content to be read, which allows for advanced management and detection of traffic patterns in a network in an Intrusion Detection System. Packet header and body contents are compared to rules which filter malware, viruses, worms, and vulnerabilities, and in the case of an IDS, flag the interaction and prevent the host from communicating.

Examples of DPI monitors include Bro/Zeek, Snort, and Suricata. DPI is especially useful in the case of enterprise laptops, which leave the work network and may bring in malware, which could spread to the rest of the network if left unchecked. DPI can use signature detection, which determines if a certain known attack is being used, which is automatically filtered from the network. DPI can also check for protocol anomalies, only allowing traffic through that fits constrained patterns that are appropriate for the declared protocols. Anomaly detection uses more general rules than exact signature matching which allows for the detection of unknown attacks. Using these techniques, DPI can reliably prevent known attacks and can determine if a packet might be malicious based on a statistical or rule-based approach. One weakness of DPI is that it is unable to beat encryption, which ultimately means that packets that are encrypted, such as HTTPS packets, are unable to be intercepted and read, which means that a careful attacker would be able to slip data through the network as long as it was being used as if the host was creating HTTPS traffic. If the rules are configured too conservatively, attacks may be missed, and if the rules are too strict, large amounts of legitimate traffic could be blocked, which would create a lot of problems for an enterprise network. To deploy a DPI, traffic must be connected to a server, either by routing the entire network through the monitor, or using a splitter. In general, DPI is relatively inexpensive, as signature matching is easy, but the more complicated the inspection processing, the slower network can be, but adding additional processing power only requires splitting the work, as each packet is inspected individually. A malicious attacker could attempt to overload the monitor, especially if the network is split rather than routed through. DPI can prevent an attacker from sending malware over the network and can prevent the sending of command and control messages, depending on the protocol and content. Ultimately, the rules and processing power of a DPI monitor greatly influence how useful and accurate it will be, while constantly balancing false positive and false negatives.

Network Flow Detection

Rather than inspecting individual packets that pass through the network, network flow monitoring evaluates the information flow between hosts on the network in search of anomalous traffic patterns, or for anomalous communication. A traditional packet-based inspection would not detect if a single host on the network was using the vast majority of the bandwidth, or a significantly greater amount than its typical usage. Similarly, if a host does not usually communicate with certain hosts (such as other user end-hosts rather than a server), a network flow monitor might be able to detect that potentially suspicious activity. Essentially, network flow technologies like NetFlow or JFlow collect data about who is communicating on the network, with whom, and what they are doing in an aggregated format (called a flow). Network flow monitoring can be used at a gateway router to detect connections going in and out of the enterprise network or within the network to detect communications between hosts. Examples of companies that have network flow analysis software include Cisco, sFlow, Juniper, SolarWinds, ManageEngine, Paessler PRTG, nProbe, and there exist many others. Compared to packet monitoring, network flow administration tools are better at detecting changes in network traffic that might correspond to newly infected malicious hosts, as they are likely to alter the information flow across the network. Highly sophisticated malware may disguise its network traffic to conform with normal traffic, but that severely limits the ability of the malware to communicate or for it to DDoS hosts inside or outside of the network. A weakness of flow-based network monitoring is that it is unable to detect vulnerabilities that rely on normal traffic like HTTPS communication or other forms of basic communication like DNS lookups. Attempting to increase sensitivity to changes in network traffic to detect the minor changes, in this case, yields many false positives, which essentially makes the data useless. In a given network, almost all traffic will be benign, and the volume of malicious traffic is generally very low, so the higher rate of false positives, the more likely for a legitimate alert to be buried. Some advanced monitoring tools address this problem by making use of artificial intelligence to help discriminate malicious traffic, but that only partially mitigates the problem. To deploy a network flow monitor, similarly to DPI, the traffic must be routed through a server used to sniff the traffic, and thus a monitor will either be positioned between hosts, which forces a star-type topology or between the network and the outer internet at a gateway router. Network flow monitors face another trade-off, the deeper and more complex the inspection of traffic flow becomes, the greater the chance of detecting malicious traffic, but this comes generally with significant performance costs, as every network interaction must be analyzed. Communication bandwidth might not be affected, as the packets can be inspected while observing traffic rather than obstructing the flow until it is verified, but the number of resources spent increases with the number of hosts and communication volume on the network. The constrained network topology requires a star-shaped network, as each end host and server must route through the monitor in order to allow the monitor to read the traffic, and this creates a lot of overhead for scalability. Thus, with the increasing scale, it becomes more likely that inspection must be more cursory to reduce performance (and thus, resource) costs, and that network traffic will receive more cursory analysis.

Software-Defined Networks

Software-defined networking gives control of the configuration to programmers rather than leaving it up to hardware and networking protocols to control the network. SDN gives a more central control such that there is software running to control and direct the hardware. The main controller will connect to all the hardware like switches through protocols like OpenFlow to send rules to all the hardware. An example of one of these rules could be to drop packers with a specific IP address or to send that packet to another switch. This makes so SDN can be changed easily[1].

One of the main strengths of SDN is being able to create complicated rule sets for a network and have it be dynamically controlled by software. This addresses one of the main reasons SDN has become so popular, old static architecture is unsuitable for modern dynamic computing and

[1] https://pdfs.semanticscholar.org/6f91/1454d11617134afa20144b73d391c6942da6.pdf

data management of large data centers and enterprise companies. For example in today's big data world companies require massive parallel processing on thousands of servers. Suddenly scaling up networking is fair easier with dynamic solutions rather than static ones that cannot be easily adjusted.

Software-defined networks are not perfect as they introduce a single point of failure. If the controller was to be attacked or go down unlike traditional networks the whole network may also go down. One of the other main issues is that it can be expensive to implement if a company needs to upgrade to hardware that supports OpenFlow, there are still a lot of legacy devices that don't that will need to be upgraded. Also, there can be issues on the placement of the main controller. If the network is too large finding an optimal controller placement can be difficult.

If a defender was to deploy an SDN they would need to figure out two main things a network controller and load balancing. They would need to determine if the overhead of doing this actually has a significant return for why they are trying to do it. Then they will need to make sure that their systems are secure if a hacker was to get control of the main controller than the whole network would be compromised. Once they figure out this main issue they can start deploying and using OpenFlow on their hardware.

SDN can allow for cloud based middlebox that is able to monitor residential networks through systems like TLSDeputy that force all TSL connections to be made through verified TLS servers. This type of approach using SDN allows for the easy scalability and security needed for something like TLSDeputy to be implemented. This type of system can be extended to many protocols promoting the security of many people who do not know what threats they face due to SDN[2].

INFRASTRUCTURE BUILDING

For this project, we decided to use NTP as our covert channel for command and control, so in order to generate cover traffic, we created a legitimate NTP server. In order to do this, we installed 'ntp' on host3 (the server) and host5 (the client), and configured both to use 10.4.8.134 as their only NTP server. We also configured the minpoll and maxpoll settings to 3 and 6, respectively, in order to make the polling frequent enough for us to get regular legitimate traffic. NTP servers start off polling at a minimum polling rate, equal to $2^{minpoll}$, then slowly back off on polling frequency as trust is gained until they get to $2^{maxpoll}$ (RFC 5905), so for our setup, the polling rate started at 2^3=8 seconds, and then quickly slowed down to 2^6=64 seconds. The default rates for minpoll and maxpoll are 6 (64 seconds) and 10 (~17 minutes), respectively (RFC 5905). The specific commands and config changes required to implement this is listed in the Appendix.

[2] https://web.cs.wpi.edu/~cshue/research/cns16.pdf

For our defense, we planned on simulating a hardware-based IDS/IPS that would intercept packets on their way into a subnetwork, so we prepared for this in the Infrastructure Building phase by routing all traffic to the infected computer (host5) through a single gateway (host4) by assigning host 4 and 5 IP addresses in a /26 subnet and hosts 3, 4 and 6 in a different /26 subnet, while adding a path between the subnets through host 4. Hosts 1 and 2 were not in either of the /26 subnets because we did not intend to capture the communication with them, so they could communicate directly. The specific details of implementing both parts of the infrastructure are listed in the Appendix.

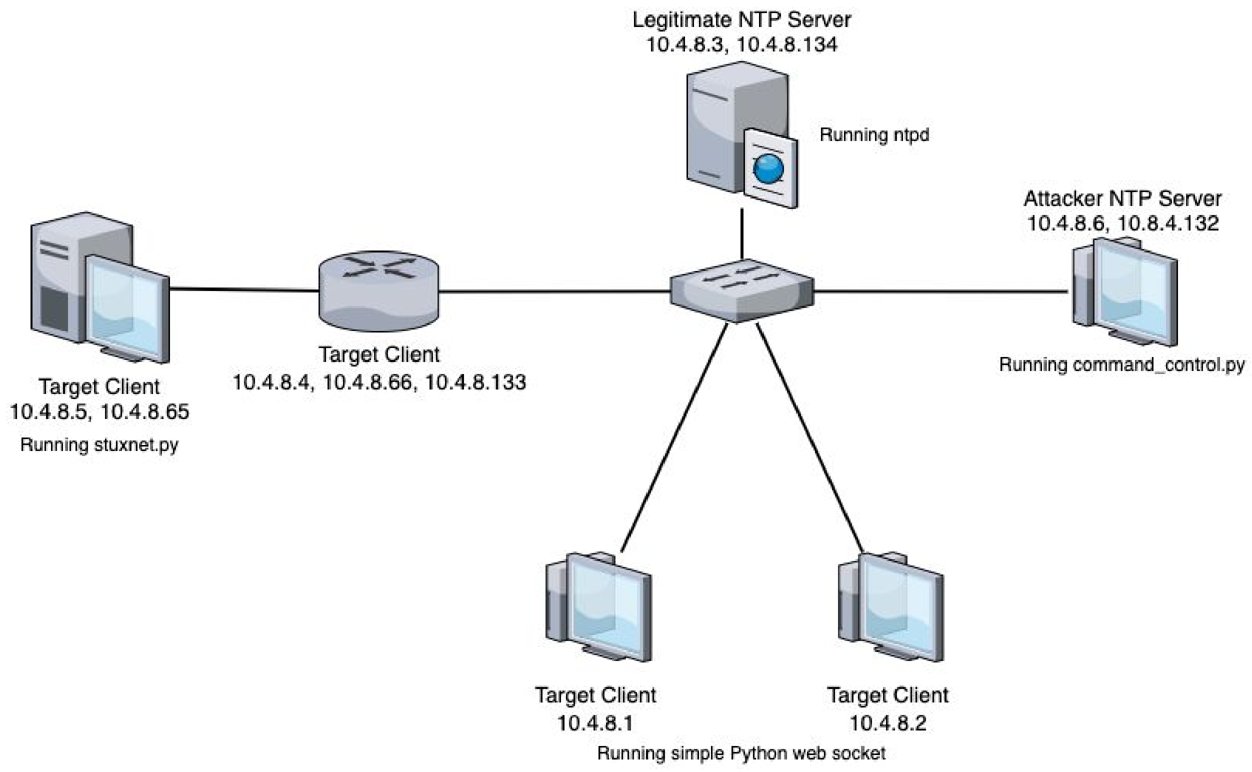


Figure 1: Initial Network Setup

ATTACK

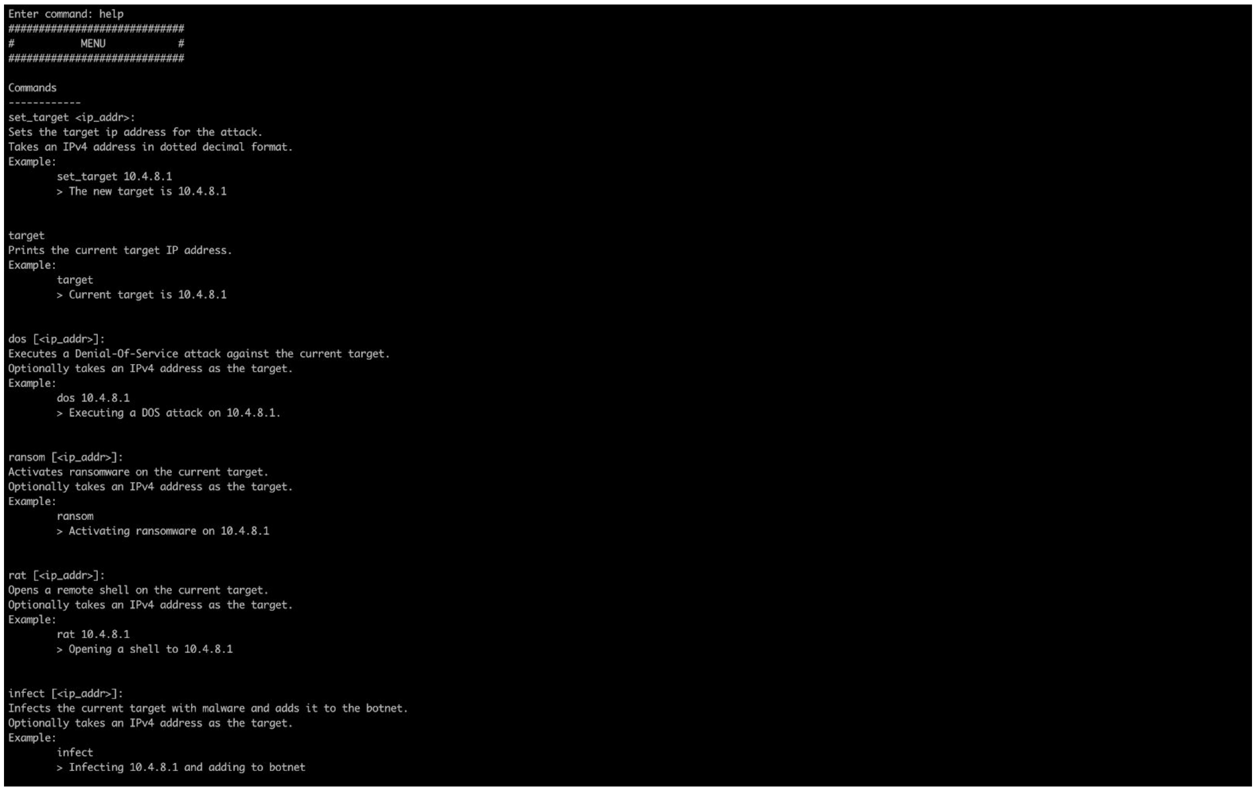
In order to use NTP as a covert channel, we first investigated how the protocol works, and what the packet fields are. We discovered a 32-bit field called ‘reference ID,’ that allows for fairly arbitrary input with little impact on the functioning of the protocol. The reference ID field is used differently depending on the value of another field called the ‘stratum.’ The stratum indicates how many hops away an NTP server is from the original clock source. At stratum 0 (the server connected directly to the clock source), the reference ID field is used as a ‘kiss-of-death’ code, which is used for debugging and monitoring. At stratum 1, the field is used as a 4-byte series of ASCII characters which identify the clock source. IANA maintains an authoritative list of these clock source IDs, but any four characters can be used, although strings starting with X are reserved for experimentation. Above stratum 1, the field is used as the reference identifier for the NTP servers, and frequently contains the IP address of server's upstream NTP server. This is the way that we used the field. In order to control our bot (host5), we sent two packets using different reference IDs. The first packet contained the target IP address for an attack, and the second packet contained a 4-character string indicating what type of attack to do. The string was converted into bytes, then formatted using dotted decimal (IPv4) notation to look like an IP address. All command strings began with '\0' to identify themselves as commands. The malware running on host5 would then sniff the incoming NTP packets, and run a preset function based on the command. The list of commands we implemented is listed in Table 1 below.

Command Code	Attack
'\0DOS'	Perform a Denial-Of-Service attack on the target
'\0RNS'	Activate Ransomware on the target
'\0RAT'	Remotely access, or open a remote shell, on the target
'\0IFT'	Infect the target with malware, adding it to the botnet

Table 1: Command Codes

In order to implement this, we created two pieces of software: the malware program to run on the bot, called Stuxnet, and the control program to run on the attacker, called NTP Command and Control Computer Program (NTP-CCCP). Our Stuxnet malware, inspired in name, but not function, by the original Stuxnet worm, used kamene (a Python 3 version of Scapy) to sniff all incoming NTP traffic, then pulled out the reference ID field if it came from the attacker's IP address. Once obtaining the reference ID, it first checks if the first byte is '\0.' If it is not, stuxnet updates its stored target to the IP address in the field. It the first byte is '\0,' stuxnet interprets the command code, and calls the proper function to attack the stored target IP. Since this is simply a simulation, we didn't actually program the attacks though. Instead, we just created a tcp web socket between the bot and the target and sent a simple message indicating that it was under a specific type of attack. The setup required to run this system is very simple, since the initial infection was assumed to have already happened. The server.py web socket needs to be placed on the target VMs and run. In a real system this wouldn't be necessary; it is only used to simulate the attack and demonstrate its success. On the bot VM, kamene needs to be installed, and the stuxnet.py and client.py scripts need to be present. The stuxnet.py script must be running. Finally, on the attacker VM, the attacker simply needs to install kamene and run command_control.py. Detailed instructions for this setup can be found in the Appendix.

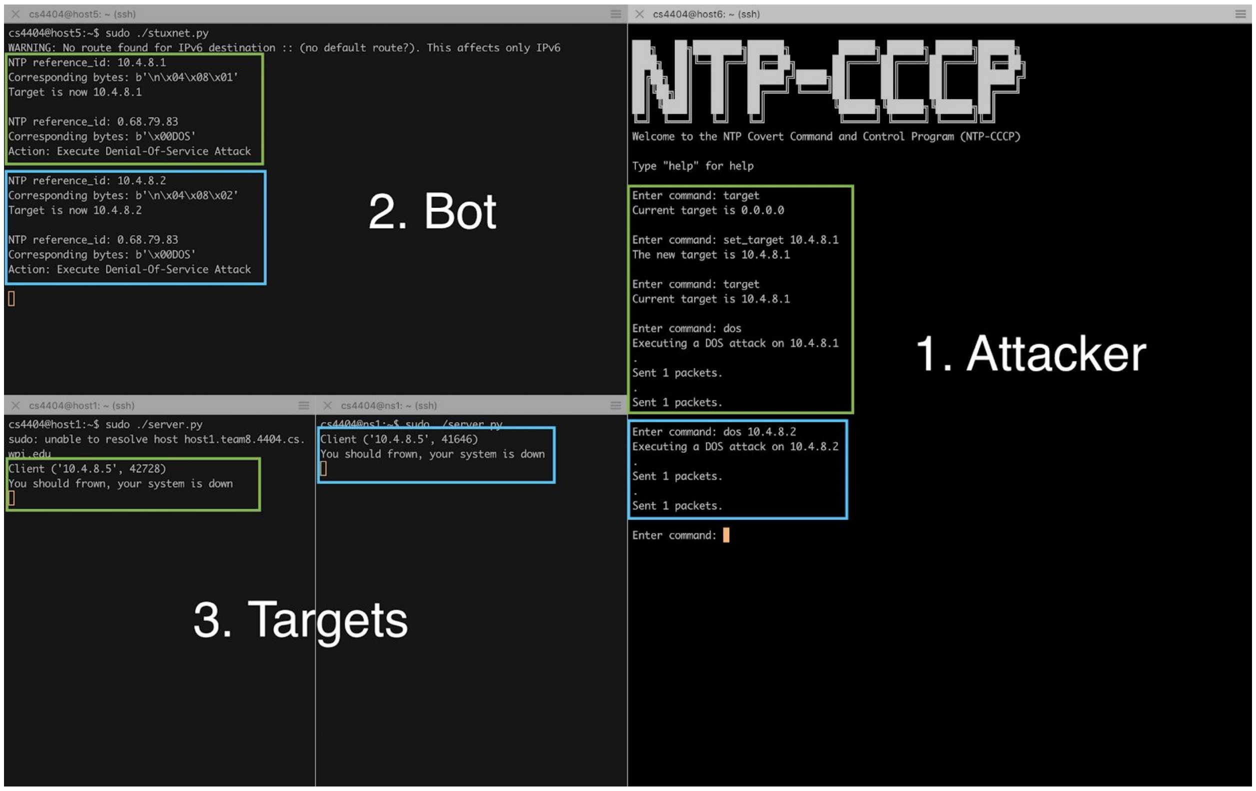
For NTP-CCCP, we designed a framework to allow easy control over the bot. We designed the framework to be easy to use, so it includes a menu detailing all of its capabilities, accessible with the 'help' command. This help menu is shown in Figure 2 below.



```
Enter command: help
#############################
#          MENU            #
#############################

Commands
------------
set_target <ip_addr>:
Sets the target ip address for the attack.
Takes an IPv4 address in dotted decimal format.
Example:
        set_target 10.4.8.1
        > The new target is 10.4.8.1


target
Prints the current target IP address.
Example:
        target
        > Current target is 10.4.8.1


dos [<ip_addr>]:
Executes a Denial-Of-Service attack against the current target.
Optionally takes an IPv4 address as the target.
Example:
        dos 10.4.8.1
        > Executing a DOS attack on 10.4.8.1.


ransom [<ip_addr>]:
Activates ransomware on the current target.
Optionally takes an IPv4 address as the target.
Example:
        ransom
        > Activating ransomware on 10.4.8.1


rat [<ip_addr>]:
Opens a remote shell on the current target.
Optionally takes an IPv4 address as the target.
Example:
        rat 10.4.8.1
        > Opening a shell to 10.4.8.1


infect [<ip_addr>]:
Infects the current target with malware and adds it to the botnet.
Optionally takes an IPv4 address as the target.
Example:
        infect
        > Infecting 10.4.8.1 and adding to botnet
```

Figure 2: NTP-CCCP Help Menu

The set of commands in NTP-CCCP allows users to view the currently selected target or change it to an IPv4 address of their choice. It also includes a command for each of our 4 attacks. The screenshot below demonstrates how to use the framework, as well as what it looks like on the bot and the target when NTP-CCCP successfully contacts the Stuxnet malware.



```
cs4404@host5:~$ sudo ./stuxnet.py
WARNING: No route found for IPv6 destination :: (no default route?). This affects only IPv6
NTP reference_id: 10.4.8.1
Corresponding bytes: b'\n\x04\x08\x01'
Target is now 10.4.8.1

NTP reference_id: 0.68.79.83
Corresponding bytes: b'\x00DOS'
Action: Execute Denial-Of-Service Attack
NTP reference_id: 10.4.8.2
Corresponding bytes: b'\n\x04\x08\x02'
Target is now 10.4.8.2

NTP reference_id: 0.68.79.83
Corresponding bytes: b'\x00DOS'
Action: Execute Denial-Of-Service Attack
```

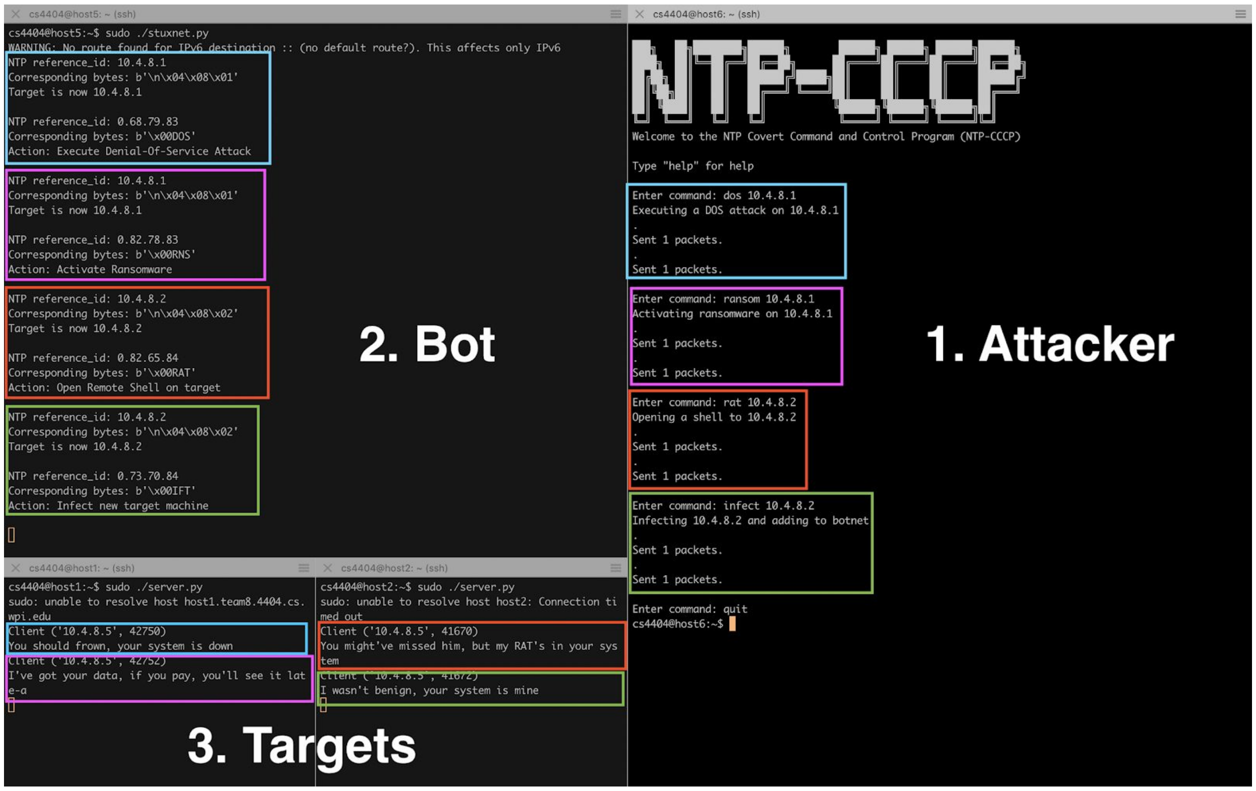
```
NTP-CCCP
Welcome to the NTP Covert Command and Control Program (NTP-CCCP)

Type "help" for help
Enter command: target
Current target is 0.0.0.0

Enter command: set_target 10.4.8.1
The new target is 10.4.8.1

Enter command: target
Current target is 10.4.8.1

Enter command: dos
Executing a DOS attack on 10.4.8.1
.
Sent 1 packets.
.
Sent 1 packets.
Enter command: dos 10.4.8.2
Executing a DOS attack on 10.4.8.2
.
Sent 1 packets.
.
Sent 1 packets.

Enter command:
```

```
cs4404@host1:~$ sudo ./server.py
sudo: unable to resolve host host1.team8.4404.cs.
wpi.edu
Client ('10.4.8.5', 42728)
You should frown, your system is down
```

```
cs4404@ns1:~$ sudo ./server.py
Client ('10.4.8.5', 41646)
You should frown, your system is down
```

Figure 3: Basic Use of NTP-CCCP

In Figure 3, the terminal pane on the right shows an attacker (host6) using NTP-CCCP to issue commands to a bot. The output of the bot (host5) is shown in the top-left pane of the terminal. The resulting attack on the target (10.4.8.{1, 2}) is shown in the bottom-left terminal panes. The sections marked with a green box show how an attacker can use a stored target IP to execute a DOS attack. The target field defaults to 0.0.0.0, so first the attacker uses the 'set_target' command to set the target IP to 10.4.8.1, and uses the 'target' command to verify it. Then the attacker simply has to use the 'dos' command to execute the attack. In the top-left pane, you can see the bot receives 2 NTP packets with reference IDs of 10.4.8.1 and 0.64.79.83 ('\0DOS'). It is not shown in the screenshot, but NTP-CCCP waits 8 seconds between sending the first packet and sending the second packet, so that it remains in the allowed polling frequency range of NTP. In the bottom-right panes, you can see a message on the target indicating that it is under attack by denial of service. You can also skip the set_target step and assign the target directly when you call the 'dos' command, as shown by the areas marked with a blue box. The final screenshot below demonstrates the use of all 4 different attack types, each marked by a different colored box.



```
cs4404@host5:~$ sudo ./stuxnet.py
WARNING: No route found for IPv6 destination :: (no default route?). This affects only IPv6
NTP reference_id: 10.4.8.1
Corresponding bytes: b'\n\x04\x08\x01'
Target is now 10.4.8.1

NTP reference_id: 0.68.79.83
Corresponding bytes: b'\x00DOS'
Action: Execute Denial-Of-Service Attack
NTP reference_id: 10.4.8.1
Corresponding bytes: b'\n\x04\x08\x01'
Target is now 10.4.8.1

NTP reference_id: 0.82.78.83
Corresponding bytes: b'\x00RNS'
Action: Activate Ransomware
NTP reference_id: 10.4.8.2
Corresponding bytes: b'\n\x04\x08\x02'
Target is now 10.4.8.2

NTP reference_id: 0.82.65.84
Corresponding bytes: b'\x00RAT'
Action: Open Remote Shell on target
NTP reference_id: 10.4.8.2
Corresponding bytes: b'\n\x04\x08\x02'
Target is now 10.4.8.2

NTP reference_id: 0.73.70.84
Corresponding bytes: b'\x00IFT'
Action: Infect new target machine
```

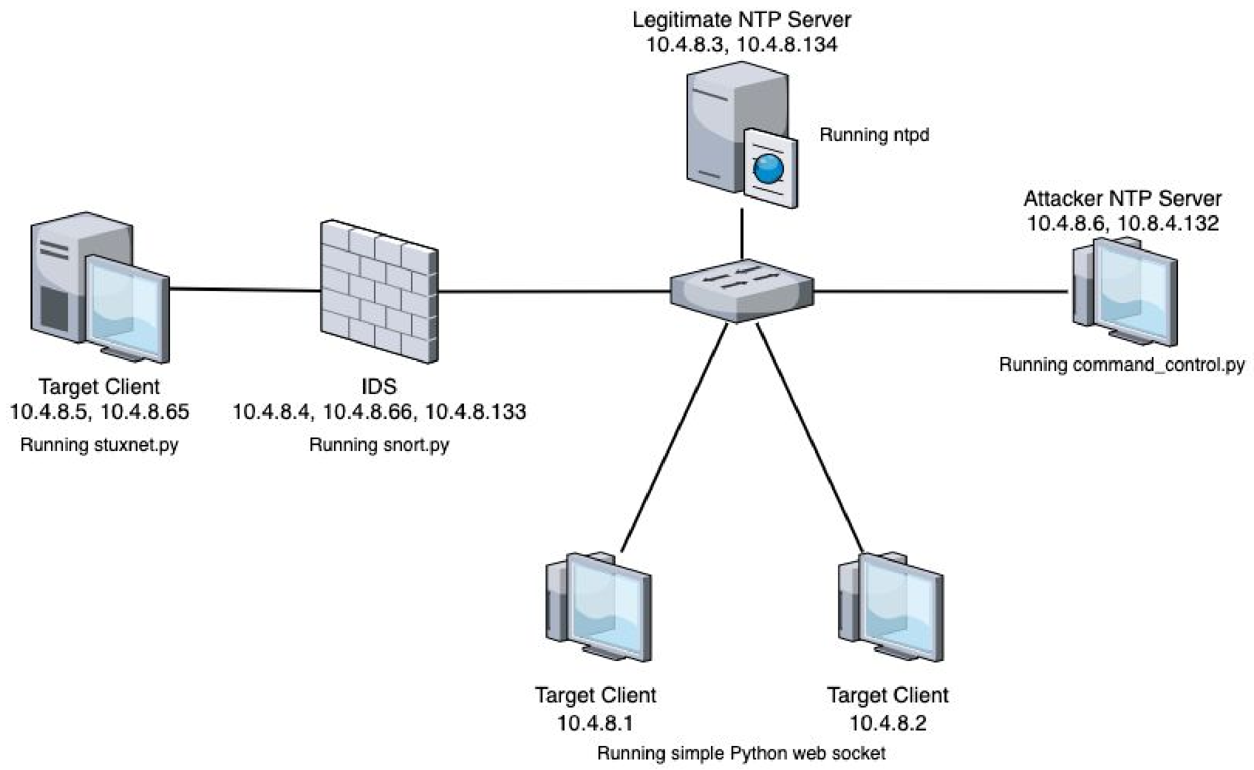
```
Welcome to the NTP Covert Command and Control Program (NTP-CCCP)

Type "help" for help

Enter command: dos 10.4.8.1
Executing a DOS attack on 10.4.8.1
.
Sent 1 packets.
.
Sent 1 packets.
Enter command: ransom 10.4.8.1
Activating ransomware on 10.4.8.1
.
Sent 1 packets.
.
Sent 1 packets.
Enter command: rat 10.4.8.2
Opening a shell to 10.4.8.2
.
Sent 1 packets.
.
Sent 1 packets.
Enter command: infect 10.4.8.2
Infecting 10.4.8.2 and adding to botnet
.
Sent 1 packets.
.
Sent 1 packets.

Enter command: quit
cs4404@host6:~$
```

```
cs4404@host1:~$ sudo ./server.py
sudo: unable to resolve host host1.team8.4404.cs.
wpi.edu
Client ('10.4.8.5', 42750)
You should frown, your system is down
Client ('10.4.8.5', 42752)
I've got your data, if you pay, you'll see it lat
e-a
```

```
cs4404@host2:~$ sudo ./server.py
sudo: unable to resolve host host2: Connection ti
med out
Client ('10.4.8.5', 41670)
You might've missed him, but my RAT's in your sys
tem
Client ('10.4.8.5', 41672)
I wasn't benign, your system is mine
```

Figure 4: Different Attacks in NTP-CCCP

DEFENSE



To defend against using NTP being used as a covert channel we found that the most realistic way to detect an attack is through verifying the reference ID field and the polling field. Reference ID is a 32-bit field used to identify the source of the NTP packet and the polling field is used to set the interval between when NTP packets are sent in the format 2^x. Such that there is a minimum polling value which is usually 3 which results in NTP packets being sent ever 2^3 seconds and a max polling value which is usually 2^{17} seconds. With a longer and stable connection, the NTP protocol will gradually increase the polling value until it reaches the max polling value, it is rare for a polling value to decline[3].

We identified and used two main ways to verify if an NTP packet is legitimate. We can check if the reference ID contains the IP address of a known NTP server and if the NTP packets are being sent at intervals that are consistent with polling intervals.

We decided to use deep packet inspection on a proxy server through mainly using the Python modules Kamene and Netfilterqueue in a script called ‘snort.py’. These tools allowed users to check each NTP packet and determine if it should drop the packet or to go to the client. The script creates many dictionaries that store information about each NTP packet by source IP

[3] https://tools.ietf.org/html/rfc5905

address. This allows us to tell what servers are sending the most NTP data and allows us to instantly differentiate known bad servers from good servers.

The script stores a list of known good NTP servers, if an NTP packet is received that does not match one of the servers listed, its reference ID will not match and an alert will be generated. The script also keeps track of the time each NTP packet was received from each IP address. If the time between the last packet and the current packet does not match the normal polling interval of 2^x then that packet will fail that check. We are able to calculate this by taking the log2 of the time difference, if that value is not close to an integer by more than a given threshold (we found the value of 0.35 through experimentation) then we know that the packet does not fit within the acceptable range of polling intervals and therefore must be invalid. As the intervals start to increase, the threshold should be lowered, but because we were only using intervals with less than a minute we found 0.35 to be acceptable. The final check uses the previous value of the polling interval by storing each value, and checking if the value decreases over time. In a sustained connection, the polling interval should monotonically increase over time. A decrease in polling interval is abnormal and thus is flagged as suspicious activity by our monitor.

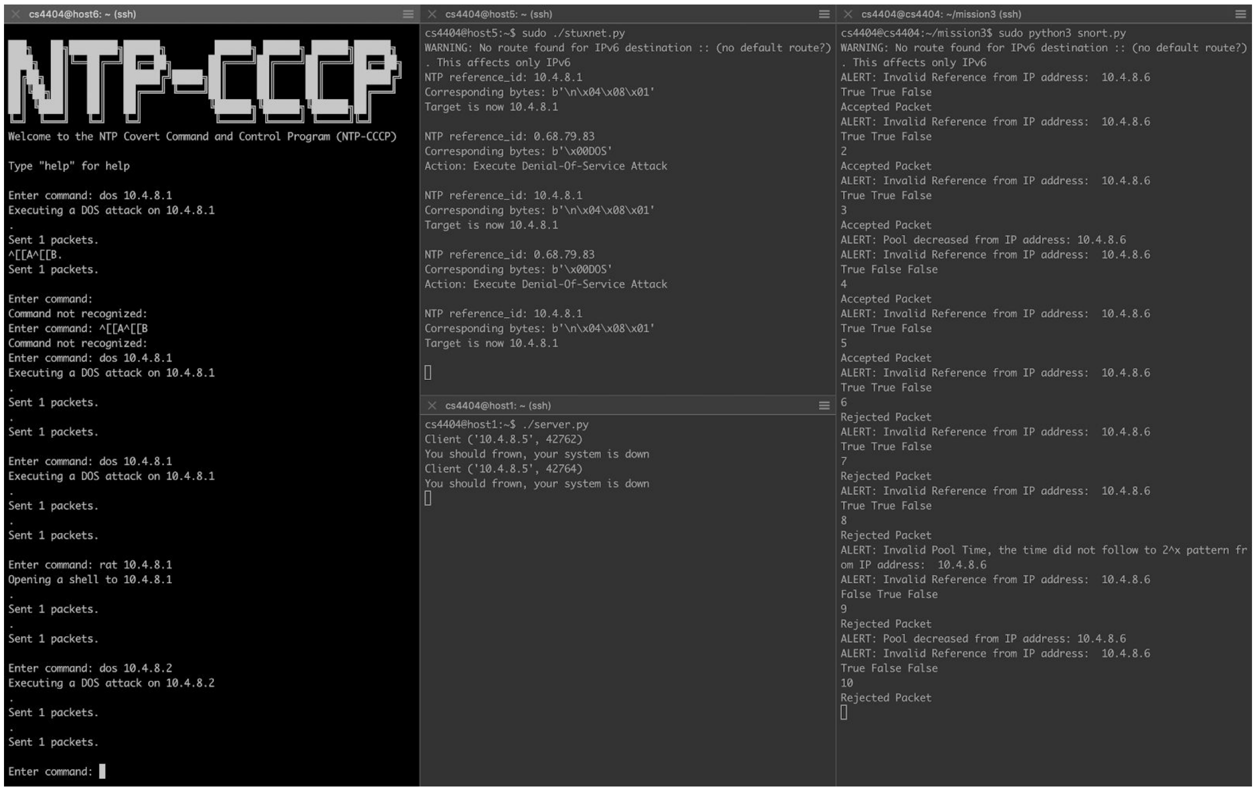
If a packet fails any of the tests source IP address of that packet is recorded. In order to account for error, no packets will be dropped until there are five recorded failed packets from that IP address. A packet will only be dropped if it fails any of the tests and that IP has more than five failed packets. A legitimate user is unlikely to fail any single test, while a malicious user is likely to fail at least one test many times as it communicates. However, even a single failure is likely cause enough to check the source of the failure for malware.

We tested our system running our malicious script and an NTP server we set up and the script was able to allow legitimate traffic while dropping the malicious traffic that did not pass inspection.

The only time there would be a false positive with our monitoring is if the reference ID generated by the command and control script is somehow the same as an IP address of one of the verified NTP servers.

Instructions to setup our defense can be found in the Appendix.

Defense blocking malicious traffic and allowing valid NTP traffic.



```
cs4404@host6: ~ (ssh)

NTP-CCCP
Welcome to the NTP Covert Command and Control Program (NTP-CCCP)

Type "help" for help

Enter command: dos 10.4.8.1
Executing a DOS attack on 10.4.8.1
.
Sent 1 packets.
^[[A^[[B.
Sent 1 packets.

Enter command:
Command not recognized:
Enter command: ^[[A^[[B
Command not recognized:
Enter command: dos 10.4.8.1
Executing a DOS attack on 10.4.8.1
.
Sent 1 packets.
.
Sent 1 packets.

Enter command: dos 10.4.8.1
Executing a DOS attack on 10.4.8.1
.
Sent 1 packets.
.
Sent 1 packets.

Enter command: rat 10.4.8.1
Opening a shell to 10.4.8.1
.
Sent 1 packets.
.
Sent 1 packets.

Enter command: dos 10.4.8.2
Executing a DOS attack on 10.4.8.2
.
Sent 1 packets.
.
Sent 1 packets.

Enter command:
```

```
cs4404@host5: ~ (ssh)
cs4404@host5:~$ sudo ./stuxnet.py
WARNING: No route found for IPv6 destination :: (no default route?)
. This affects only IPv6
NTP reference_id: 10.4.8.1
Corresponding bytes: b'\n\x04\x08\x01'
Target is now 10.4.8.1

NTP reference_id: 0.68.79.83
Corresponding bytes: b'\x00DOS'
Action: Execute Denial-Of-Service Attack

NTP reference_id: 10.4.8.1
Corresponding bytes: b'\n\x04\x08\x01'
Target is now 10.4.8.1

NTP reference_id: 0.68.79.83
Corresponding bytes: b'\x00DOS'
Action: Execute Denial-Of-Service Attack

NTP reference_id: 10.4.8.1
Corresponding bytes: b'\n\x04\x08\x01'
Target is now 10.4.8.1
```

```
cs4404@host1: ~ (ssh)
cs4404@host1:~$ ./server.py
Client ('10.4.8.5', 42762)
You should frown, your system is down
Client ('10.4.8.5', 42764)
You should frown, your system is down
```

```
cs4404@cs4404: ~/mission3 (ssh)
cs4404@cs4404:~/mission3$ sudo python3 snort.py
WARNING: No route found for IPv6 destination :: (no default route?)
. This affects only IPv6
ALERT: Invalid Reference from IP address:  10.4.8.6
True True False
Accepted Packet
ALERT: Invalid Reference from IP address:  10.4.8.6
True True False
2
Accepted Packet
ALERT: Invalid Reference from IP address:  10.4.8.6
True True False
3
Accepted Packet
ALERT: Pool decreased from IP address: 10.4.8.6
ALERT: Invalid Reference from IP address:  10.4.8.6
True False False
4
Accepted Packet
ALERT: Invalid Reference from IP address:  10.4.8.6
True True False
5
Accepted Packet
ALERT: Invalid Reference from IP address:  10.4.8.6
True True False
6
Rejected Packet
ALERT: Invalid Reference from IP address:  10.4.8.6
True True False
7
Rejected Packet
ALERT: Invalid Reference from IP address:  10.4.8.6
True True False
8
Rejected Packet
ALERT: Invalid Pool Time, the time did not follow to 2^x pattern fr
om IP address:  10.4.8.6
ALERT: Invalid Reference from IP address:  10.4.8.6
False True False
9
Rejected Packet
ALERT: Pool decreased from IP address: 10.4.8.6
ALERT: Invalid Reference from IP address:  10.4.8.6
True False False
10
Rejected Packet
```

CONCLUSION

From this mission we were able to learn about how easy it is to send data through covert channels to communicate with a botnet. We were able to apply techniques that we learned from researching packet inspection tools to design and implement our own. By doing this we learned how insecure the internet can be; even some benign field in a UDP packet could contain a command to launch a distributed attack across many systems. From our attack, we learned how easy covert command and control is to implement, as we were able to build a program that will generate an obfuscated packet with malicious data and send it to a target to launch an attack. With simple modifications, a script like this could send commands to thousands of machines.

CITATIONS

Network Time Protocol Version 4: Protocol and Algorithms Specification. (2010, June). Retrieved from https://tools.ietf.org/html/rfc5905.

APPENDIX

How to run our simulation

Infrastructure Setup

Copy Files to Hosts

- scp files to respective hosts.
- Each host has its own directory.
- Copy kamene-master to host 4,5, and 6

Setting up IP Aliasing & Setup

- In each interface file the IP address must be changed to match the IP address of the VMs you're using.
- On the listed VMs in the host file you can find the interfaces file.

```
~~~~~~ host 2 ~~~~~~
cp interfaces /etc/network/interfaces
sudo ifdown eth0 && sudo ifup eth0

~~~~~~ host 3 ~~~~~~
cp interfaces /etc/network/interfaces
sudo ifdown eth0 && sudo ifup eth0
sudo apt install ntp
cp ntp.conf /etc/ntp.conf                # edit line 22 to match your IP address
sudo service ntp restart

~~~~~~ host 4 ~~~~~~
cp interfaces /etc/network/interfaces
sudo ifdown eth0 && sudo ifup eth0
sudo sysctl net.ipv4.ip_forward=1
#in kamene-master directory
sudo python3 setup.py install
```

```
cd ..
cd NetfilterQueue-0.8.1
sudo apt install python3-dev build-essential libnetfilter-queue-dev
sudo python3 setup.py install

~~~~~~ host 5 ~~~~~~
cp interfaces /etc/network/interfaces
sudo ifdown eth0 && sudo ifup eth0
sudo sysctl net.ipv4.ip_forward=1
sudo apt install ntp
cp ntp.conf /etc/ntp.conf                  # edit line 18 to match your IP address
sudo service ntp restart
#in kamene-master directory
sudo python3 setup.py install

~~~~~~ host 6 ~~~~~~
cp interfaces /etc/network/interfaces
sudo ifdown eth0 && sudo ifup eth0
sudo sysctl net.ipv4.ip_forward=1
#in kamene-master directory
sudo python3 setup.py install
```

Attack

The attack requires four terminals. To hosts 1,2,5,6

~~~~~~ host 1 ~~~~~~

- Launch server that infected host will communicate with.
    - ./server.py

~~~~~~ host 2 ~~~~~~

- Launch server that infected host will communicate with.
    - ./server.py

~~~~~~ host 5 ~~~~~~

- Launch malware on infected host that read incoming NTP traffic for commands.
    - ./stuxnet.py

~~~~~~ host 6 ~~~~~~

- Launch attack client that communicates with infected host.
    - ./command_control.py

Defense

~~~~~~ host 4 ~~~~~~
~~~~~~

- Launch script that will intercept every NTP packet going to host 6 and inspect it. You must add the ip address of the NTP server so the script will know its a valid server. Right now the only approved server is 0.0.0.0
    - ./snort.py

Readme

Host 1

- interfaces
    - Copy of what /etc/network/interfaces should be.
- server.py
    - Simple web socket used to show command and control infected host

Host 2

- interfaces
    - Copy of what /etc/network/interfaces should be.
- server.py
    - Simple web socket used to show command and control infected host

Host 3

- interfaces
    - Copy of what /etc/network/interfaces should be.
- ntp.conf
    - Copy of ntp server config /etc/ntp.conf

Host 4

- NetfilterQueue-0.8.1
    - Module to bring packet to python script
- kamene-master
    - Scapy for python3
- interfaces
    - Copy of what /etc/network/interfaces should be.
- snort.py
    - NTP packet inspector that filters malicious NTP packets

Host 5

- kamene-master
    - Scapy for python3
- interfaces

    - Copy of what /etc/network/interfaces should be.
- ntp.conf
    - Copy of ntp config /etc/ntp.conf
- stuxnet.py
    - Python script that interprets NTP packets and launches attacks.
- client.py
    - Simple web socket that “attacks” other machines

Host 6

- kamene-master
    - Scapy for python3
- interfaces
    - Copy of what /etc/network/interfaces should be.
- command_control.py
    - Python script that allows the hacker to send commands to their infected machines through NTP packets.