Packets, Policies, and Permissions: Towards Lightweight Network Capabilities

CS4404 - A16: Final Project

Lauren Baker (lmbaker@wpi.edu)
Domenic Bozzuto (dlbozzuto@wpi.edu)
Evan Gilgenbach (evan@wpi.edu)

Abstract

This paper details the design, implementation, and evaluation of an alternative, lightweight approach to network capabilities, using commodity hardware. By using DNS to dynamically return unique IP addresses to individual clients for the same services, networks are given much more flexibility in protecting their servers from malicious traffic. Our implementation allows users to flexibly configure many different elements and aspects of the system to easily achieve their unique security goals.

Introduction

Overview

The goal of this project is to develop and implement a network system that uses capabilities to protect against attacks. In particular, this type of implementation is designed to reduce the amount of damage caused by Denial of Service (DoS) attacks.

This is done based on the proposed network with capabilities described in "On Building Inexpensive Network Capabilities". A capability refers to the ability of a client to be able to access a network or host; without the capability, the client cannot access the network or host. Many types of network capabilities have been proposed before, often with an accompanying extension to the TCP, UDP, or IP protocols. In contrast, this paper proposes a method of distributing capabilities that works with existing network stacks and clients, without requiring specialized hardware or software. This is a huge improvement over existing work in the network capability space.

To implement capabilities, the authors proposed using a NAT router that can dynamically associate IPs with protected servers, also known as "IP fluxing". These IP addresses would be created dynamically per-client in response to a DNS request for the hostname. Without a proceeding DNS request, there's now no way for a client to access a protected server. In this way, the IP address corresponding to the host name acts as the capability. If the client is determined to be malicious, it can be denied access to the network before it even has the opportunity to interact with a protected system.

Extension

The core of the extension is the concept of a human-readable "policy configuration file" that will allow system administrators to customize the behavior of the capabilities server towards different clients. For example, the configuration policy could allow the system administrator to configure specific TTLs for different devices on the network (such as a TTL of 20 for the client "10.4.2.1" and a TTL of 60 for the client "10.4.2.2"). If a client on the network is suspected to be malicious, for example, the administrator can impose more restrictions on that device. By implementing this configuration file, the network owner will have much more control over the devices on his or her network.

To provide an example of this extension, suppose the network administrator is very confident that his computer is not infected with any malware that would attack his network. The network administrator would include a specification in the policy file for his computer, “10.4.2.1”, to provide a TTL of 120 for any DNS lookup provided by the server. Because the administrator’s computer is likely not malicious, he can use longer TTLs to reduce the overhead on the DNS server.

Another person has a computer with IP address “10.4.2.2”, which has just connected to this network. Because the administrator is unsure of how this device will behave, he can add a clause to the policy file that enforces stricter TTL times on the device. A sample policy file configuration for this is illustrated below:

```
[trusted_connection]
selector = “10.4.2.1”
TTL = “120”

[risky_connection]
selector = “10.4.2.2”
TTL = “15”
```

Threat Model

One of the threat models we considered while creating this design document was a single-client, non-distributed Denial of Service attack. Once a capability is granted to such a party, their attack would not be impeded by this implementation of a capabilities system. Other components of the network will not stop unwanted traffic, so the attack would be free to continue for the duration of the Time to Live (TTL) in the DNS response. Our implementation will include logging client behavior, but it will not have any built in response to potentially unwanted client behavior, so such a client would be able to get another capability after the first expired. However, in a network with monitoring infrastructure, such malicious behavior could be identified and future DNS requests could be sent a different IP address (e.g. one that is not currently in use, or one for a honeypot) using the simple configuration system.

One threat this capabilities system will provide some protection against is scanning. Because the mappings from public IP addresses to the protected server will be changing frequently, random traffic will be less likely to be productive and IP-based hit lists will be similarly limited. If a scan did find a vulnerability, assuming the server was not yet compromised, the frequent changes in address will make it more difficult for the attacker to return and exploit what was discovered. It is also worth noting that if no capabilities have been granted at a given time, no scan of IP addresses will be successful.

Another threat that must be considered in creating this system is a distributed denial-of-service attack. With limiting on the rate at which capabilities will be given out, an attempt at a DDoS attack becomes a denial-of-capabilities attack as the DNS server receives a large volume of requests. This DoC attack would disrupt legitimate traffic, meeting the same goal of the intended DDoS attack. However, the impact might be lessened because the DNS server’s processes are lightweight and may require more traffic to overwhelm when compared to the server that is to be protected.

By allowing an network owner to change the policies enforced by the system, certain attacks can be mitigated. For example, if the network owner realizes the network is being DDoSed by a massive number of external clients, he can select a policy to deal with these external attacks. He may choose, for instance, to block all requests from particular external subnets. Giving the end user policy control allows for more customizable control of the network based on the current conditions faced by the network.

Implementation

To implement this system, 4 virtual machines will be used. The VMs will be used as follows:

1. Client
2. Client
3. Capability Server (including the DNS server and the NAT)
4. Web Server

Any software that needs to be developed for this implementation will be developed with Python. Figure 1.1 shows a high level overview of the system.

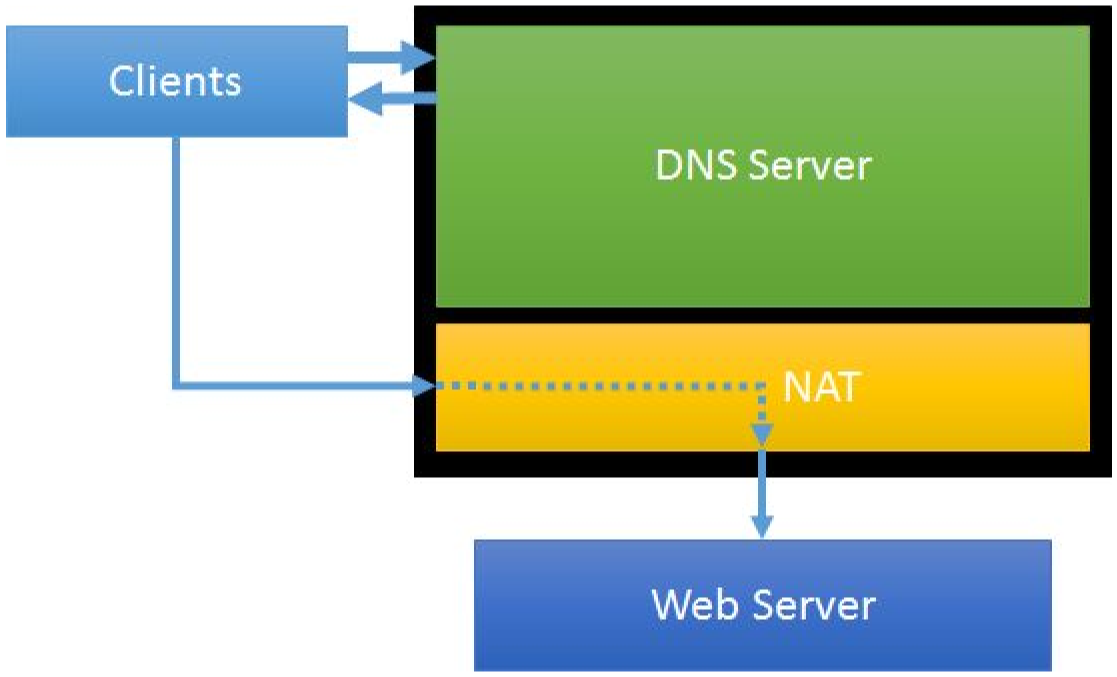

Figure 1.1: High level diagram showing the path connections will take through the system.

Implementation: Clients 1 and 2

The two client devices were implemented on a virtual machine running Ubuntu (other variants of Linux could have been used, but for this implementation Ubuntu was used). These clients should use “10.4.2.1” and “10.4.2.2” as their IP addresses, respectively. In order to successfully gauge the performance of the network, these clients must be capable of performing several functions.

Perform Legitimate Requests:

Clients will be able to make normal connections to the web server that properly use the capability system. This will be accomplished via two primary means:

- `curl -4 webserver.isp`
  This function is useful for connecting to the web server via the command line. For some tests where a scripted solution is infeasible, this command will be used to assess the functionality of the system. When the command is successful, it print the contents of the server's "index.html" to the terminal window. When the connection fails, the command will indicate the connection has failed

- `socket.gethostbyname("webserver.isp")`
  This function performs a DNS query on the specified hostname (in this case, "webserver.isp"), and returns the IP address associated with the hostname. In this implementation, a successful call of `socket.gethostbyname` should return an IP address in the range in the subnet "10.4.2.128/25", and should *never* return the IP address of the webserver itself, "10.4.2.4"

The `gethostbyname` function will primarily be used within a script called `measure_lookup_time.py.` This script will take two command line arguments specifying the number of lookups to perform and the delay (in seconds) between lookups; the script will perform the specified the number of lookups and will calculate the time necessary to perform the lookup. It should output the lookup times in a `.csv` format. For all other necessary connections to the web server, the `curl -4 webserver.isp` command will be used.

Perform Malicious Behavior

Clients must also be capable of performing behavior that attempts to exploit weaknesses in the capability system. To perform this malicious behavior, clients will use the following means:

- `curl -4 <IP ADDRESS OF KNOWN CAPABILITY>`
  `curl` also supports connecting to targets directly via IP addresses (and therefore bypassing the DNS lookup), so `curl` will be used as the terminal-variant of attempting a connection

- `socket.connect(<IP ADDRESS OF KNOWN CAPABILITY>, 80)`
  For scripted attacks against the system, the Python command `connect` will be used, which attempts to establish a TCP connection with the target IP address. Because the web server hosts a standard `http` page, port 80 will be used exclusively.

Similar to the benevolent behavior, it is important that the system has a terminal-based command and a script-based command to attempt to make connections.

Several malicious behaviors will be implemented by the clients:

- IP Scanning: Each client should be capable of issuing a script, `scan_ips.py`, that attempts to make a connection to every given IP address in a given range. For each IP address, the script will use the `socket.connect` command to attempt to establish the connection. Based on the return value from the function call, it can be determined if the connection was successful.

- Sharing Capabilities: One vulnerability of this system is that if the NAT is not properly configured, once a client has obtained a capability, it can share and distribute this capability with other clients. To test this, once a client has been returned a capability, another client will attempt to use `curl` to access that IP address.

- Using Expired Capabilities: A script called `test_ttl_exp.py` will take two command line arguments representing the number of requests to make and the delay between requests to perform. Once executed, the script will first use `gethostbyname` to obtain a capability, and then will attempt to make every proceeding connection with `connect`, using the capability obtained by the lookup. During the execution, the result of each `connect` attempt is printed to standard output. This function will verify that TTL expirations are enforced by the server.

Implementation: Capability Server

The Capability Server VM will use BIND9, a preconfigured package to run a DNS server. To intercept packets in outgoing DNS responses, the system will use the `netfilter-queue` library (NFQ). Furthermore, to modify the contents of these packets, the `scapy` library for Python will be used. Finally, to implement the NAT and redirect traffic to use NFQ, `iptables` will be used. Figure 1.2 shows a high-level architecture of how traffic will flow through the Capability Server.

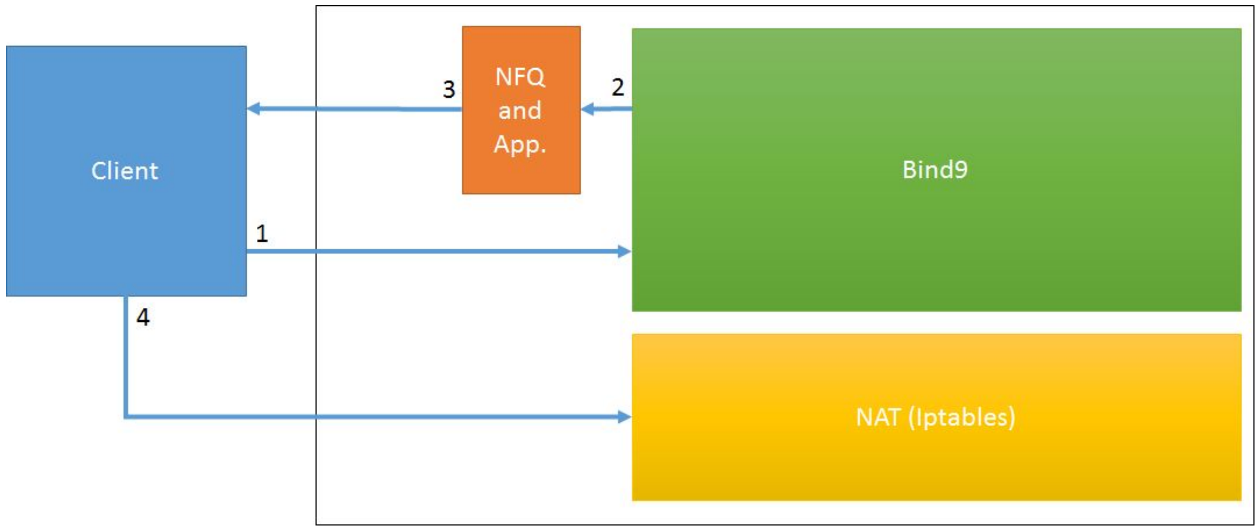

Figure 1.2: Path of traffic through the capability server.

1. The client makes a DNS lookup request to the the Bind9 server running the on the machine
2. Bind9 sends a DNS response, but this is redirected to and caught by NFQ.
3. NFQ modifies the resource data and TTL sections of the packet before sending the response back to the client
4. The client attempts to connect to the web server using the information contained in the DNS response. Because the IP address returned to the client is not the

actual IP of the web server itself, the client will be routed through the NAT (via IPtables).

Initializing the Capability Server Application

The packet interception portion of the capability server is a Python script called main.py. This script performs several actions to ensure the capability system functions correctly:

- Read the policy file:
- Establish a Capability Queue
- Intercept Packets

Each of these core functions will be explained in more detail below:

Reading the Policy File

In order to properly apply different policies to the various clients, the system must have a policy file that lists the various policies. The policy file format should be a human readable format (our implementation will use TOML, although other easily modifiable and human-readable formats are acceptable). The policy file will define certain rules, each with a selector and a set of configuration variables.

The specific actions we have planned out for our implementation are “TTL” (which specifies the TTL returned to a client in seconds), “always” (which defines whether capabilities should always be *allowed* or *denied* for a set of clients), and “strict”, which when true restricts the capability to the requesting client only. When false and by default, any client on the same AS can use the capability. An example configuration file is shown below:

```
[default]
selector = “default”
TTL = “25”

[trusted_client]
selector = “10.4.2.1:”
strict = “False”
always = “allow”
```

```
TTL = “1”

[risky_client]
selector = “10.4.2.2:”
TTL = “3600”
strict = “True”
always = “deny”
```

The contents of the policy file will be parsed into a Python dictionary, which can be used by the application at any time after the initial read.

Capability Queue

At the core of the capability server is a capability queue, which contains all of the currently active capabilities on the network. The capability queue contains several important lists of IP addresses:

- Available IP Addresses for Use as Capabilities (`availableIPs`): This list is initialized to be all of the IP addresses in the range “10.4.2.128 “- “10.4.2.255” (and leaves “10.4.2.1” - “10.4.2.127” as possible IP addresses for clients). The represent the IP addresses that can be returned to the clients as capabilities. When a capability is created, an IP address is randomly selected from this pool to be the IP address for the capability.

- Actively Used IP Addresses (`inUseIPs`): This list is initialized to be empty. As capabilities are granted, IP addresses are moved from the `availableIPs` list to the `inUseIPs` list. Similarly, as capabilities expire, the IP addresses are removed from the `inUseIPs` list and returned to the `availableIPs` list.

- Active Clients (`activeClients`): A list of clients that currently have capabilities is maintained to ensure that clients only have one capability granted to them at any given time. When a capability is created, the client’s IP address is added to this list. Similarly, when the capability expires, the client’s IP address is removed from this list.

In addition to maintaining these lists, the capability queue provides a some core functions to handle the creation and expiration of capabilities.

- New Capability: Whenever a capability is created, its expiration time is set to the current system time plus the TTL specified by the policy file. Each capability must also be provided the mapped IP address (the capability itself), and the client IP address that is paired with the capability.
- Add Capability: Whenever a new capability is added to the queue, it is placed into the queue based on its expiration time. The queue is maintained such that the first item in the queue is always the first to expire. Additionally, when a capability is added, and iptables rule is created that maps the capability IP address to the web server. The rule to accomplish this is: `{iptables} -t nat -A PREROUTING -p tcp -s {clientAddress} -d {mappedAddress} --dport 80 -j DNAT --to-destination 10.4.2.4:80`
- Check For Expirations: When this function is called, the system compares the current system time to the expiration time of the first item in the queue. If the current time is greater than the expiration time, the capability (and is associated `iptables` rule) are removed. This function is run every 500ms to ensure the queue is always up to date. This function is run within a thread in the background to avoid interfering with packet interceptions.

Intercepting Packets

The following iptables rules are used to redirect all incoming traffic to the NFQ stack:

```
iptables -I INPUT -p udp --dport 53 -j NFQUEUE --queue-num 1
iptables -I OUTPUT -p udp -j NFQUEUE --queue-num 1
```

The capability granting application intercepts every packet that flows through the “10.4.2.3” machine. Because the only outgoing traffic originating from the capability server is DNS responses, any packet that does not have a “Source” field of “10.4.2.3” can be immediately accepted.

If a the packet is DNS Response, the application traps the packet with NFQ for modification. The destination of the packet (which is the IP address of the client that performed the lookup) is extracted, and compared with the information in the policy file to determine the type of capability that should be granted to this client. If the client does not already possess a capability, it is granted a new one.

Using `scapy`, the resource data and TTL portions of the DNS response are modified to be set to the capability IP address and the policy-defined TTL, respectively. Once these values have been modified, the checksum and length of the packet are recalculated, and the packet is sent back to the client that performed the lookup.

Implementation: Web Server

The final virtual machine should run a web server. This web server will be the target of both clients. For the implementation of a rudimentary web server, `apache2` will be used. The webserver will have a very simple `index.html` file, and will have the hostname "webserver.isp". The web server does not actually need to perform any actions for clients on the system. The only requirement is that the webserver is always running, so that clients with capabilities are always able to connect. The server's html was kept short and identifiable such that it was easy to quickly to determine if the client successfully accessed the webpage.

Because the goal of this implementation is to provide a capabilities system without needing to modify the behavior of the components of the system (aside from the DNS resolution process), the web server has no relevant functionality aside from its role as a protected element of the network, and apache2 is running without any modifications to note.

2. Evaluation

In order to properly evaluate the successfulness of an implementation of the system described above, a rigid set of evaluation criteria need to be established. Before the evaluation criteria can be established, however, it is important to understand the security goals of the system. The primary security goals of the system are:

- Shift the processing load from the end web server to the DNS resolver. DNS resolvers typically have more appropriate hardware for handling large numbers of packets. This allows requests to to blocked at the DNS resolver, which would prevent large volumes of malicious traffic at the web server.

- Allow for on-the-fly updates of the system's capability policy. If the system administrator notices that system is being flooded with malicious attacks, he has the capability to change the system's capability accordingly via the configuration file to attempt to reduce the damage done by the malicious activity.

With this in mind, there are several performance measurements that can be taken to determine the effectiveness of the system.

Packet Latency

A crucial factor in determining the effectiveness of this system is the amount of time the packet is trapped by the capability server before being returned to the client. To measure the impact our capability system has on DNS response, we will compare a DNS lookup time on our network to a BIND9 server with no modifications to provide capabilities, to a test with our system modifying packets. The timing can be performed via Python's time.time() call, with a statistically significant large number of samples.

The latency will be measured on the virtual machines both before and after the implementation of the capability system. This will establish a baseline for how long the requests take on a bare-minimum BIND9 server; this can then be compared to the response time after implementing the system to determine how much time is required to perform the capability operations. The increase in latency found will be compared with

DNS lookup time results from google.com, facebook.com, and wpi.edu. It was found these sites took approximately 72 ms, 62 ms, and 45 ms, respectively; this yields an average DNS Lookup time of approximately 60 ms. To satisfy the lightweight goal of the implementation, the time taken by the capabilities system should be no more than 60 ms. In other words, adding this capabilities system should no more than double the DNS lookup time for a website with an average DNS lookup time of 60 ms.

For this test, a Python script was used on one of the two clients, while the Python script for capability granting was running, and while it was not running (in this case only the BIND9 server is used for DNS lookups). This script used Python's socket module, specifically the gethostbyname() method on the hostname of the web server. The time module was used to make a time() call before and after the gethostbyname() call, and the difference of the two times was printed in standard output. These steps were executed in a loop to complete 20 trials.

Successful Connections

For the implementation to be considered successful, clients who have obtained a capability must be able to connect to the web server. This can be measured by executing the Python script `test_ttl_exp.py` on one of the two client machines. This script, mentioned in the implementation section, takes a number of connections to make and a time delay between connections. Using Python's socket module, the web server's IP address is resolved using the gethostbyname() method, and then the specified number of connections are made using the connect() method, the IP address found in the first step, and port 80. This script was used to make 6 connections during the TTL of the capability (60 second TTL, 10 second delay between connections), to show that this client is able to connect to the web server repeatedly once obtaining permission.

Connections Without Capabilities

One key design goal of the capability system is to reduce the effectiveness of IP scanning to access the web server without performing a DNS lookup. Because the network setup we were given for this assignment doesn't have isolated networks in the same way that the test networks we used for the missions did, we assigned one network (10.4.2.128/25) to be routed through the gateway router. This was the area the test scanned to determine if any “outside” (i.e., not on the same physical network) systems would have been able to access the protected server without a capability. To be a success, during this scan there should no successful connection attempts made to the web server. The test will be conducted by one client after the other one has been granted a capability under a “strict” policy, and during the timeframe for which this capability is valid. The scan should not be able to take advantage of this capability, as its source address will not be correct.

The first step in this test was having client 2 use the curl command to access the web server, generating a capability in the process. The policy for this test was set to “strict” with a TTL of 1500 seconds to be sure to accommodate the entire scan. A Python script was run on client 1 during this TTL. The script established a socket using Python’s socket module, and attempted to establish a tcp connection to each IP address in the specified range on port 80. For the test to be a success, every attempt to establish a connection should yield the “no route to host” error.

TTL Verification

For this capability granting service to be considered successful, a client must be able to use the IP address returned to them for the entire duration of the TTL included in the response. Testing will include making connections throughout the TTL, and verifying that the connection attempts are no longer successful after the TTL has expired. This test should issue an initial attempt using gethostbyname, followed by a series of requests using the obtained IP address. The attempts will be equally spaced, and there should be at least 2 attempts before the capability expires and at least 2 after the capability has expired. This will be accomplished with the `test_ttl_exp.py` script described in the Implentation section.

Correct Policy Functionality

Any selectors added to the configuration files, as a way of denying DNS requests from certain ranges of IP addresses or for enforcing particular TTLs, must be respected. If a client meets the requirements for a policy that denies the client access to any capabilities, the connection to the web server should be blocked for that client. Similarly, if a client is selected for a policy with a particular TTL, it should be measured that the TTL expires at the correct time.

Evaluating this policy will be accomplished by monitoring the output of the Capability Server log (which lists the times at which capabilities are granted and when they expire, as well as observing the terminal output when `curl` is used to request a capability. If `curl` indicates a connection was refused or blocked, a capability was denied successfully.

To test this, a policy file was used with a policy denoted “`risky_client`”. This policy had a selector to target one of the two client IP addresses (`selector = “10.4.2.1:”`), and it had “always” set to “deny”. With this policy in effect, this client attempted to connect using `curl` and the web server hostname. For a successful test, the client should not be able to connect to the web server, which will be reflected in the output of `curl`, as the capabilities system should not create a capability for this client.

Capability Exclusivity

Evaluation will include testing whether granted capabilities can be shared more widely than the intended recipients. One client will be granted a capability, and during the TTL there will be connection attempts using this capability from clients with different IP addresses. The log of successful connections after this test should reflect that only the client to whom the capability was granted was able to use the capability.

For this test, a policy was created to cover the IPs of both clients, with the “strict” option set to true. One client used curl to connect to the web server by its hostname, gaining a capability in the process. This IP address was taken from the log of the third machine, and the other client attempted to connect using curl and this IP address. This simulates one client distributing an IP address to other machines.

Policy Switching

As updating configurations “on-the-fly” is part of the goals for the system, evaluation will include verifying the mechanism by which updates to the configuration files are implemented in outgoing capabilities. DNS responses sent after configuration files are updated must reflect the configuration changes to meet the system goals. This will be accomplished by sending a predefined group of packets through the system with no policy in place. Then, a policy should be applied and the same set of requests should be made. The set of requests should be engineered such that some of the packets will be caught by the policy. This will be tested by having the client make a request to the DNS server, which should grant a capability and allow the client to connect to the web server. Then, the policy will be updated by an administrator such that the client is no longer able to connect to the web server. The client should fail to connect to the server, which will be logged via Python on the client side.

General Requirements

In addition to the testing described above, there are also an underlying set of functionalities that must be present within the system. While most of these requirements are trivial in nature, it is important nonetheless to explicitly state them.

- Reading and writing to the policy configuration file should never fail. This will need to be programmed robustly to ensure the file is always accessible.
- Iptables updates should never fail; the software should always format valid iptables commands that can execute without causing an error.
- The DNS server should have nearly 100% uptime; there are no external factors that should cause the server to shutdown
- The web server should have nearly 100% uptime; at its peak, traffic to the web server should be moderate, so traffic volume should not inhibit the server.
- Clients should always know the IP address of the DNS server and should be able to connect to it (even if the DNS proceeds to reject their request for a capability)
- The modification of packets trapped with netfilter_queue should never fail, and invalid packets should not be created.

All of the general requirements described above can be accomplished with good coding practices and thorough testing.

Evaluation Sheet

Below is a suggest rubric for assessing the requirements outlined above. For each requirement, a description of a successful test is given. From this, the tester should select a number of trials for each element (recommended to be at least 15 trials for comprehensiveness), run the trials, and record the number of successes. If the percentage of successes is greater than a specified success rate (recommended to be around 90%), the test can be given a passing grade.

Table I: Evaluation Rubric for the Proposed Implementation

Measurement	Definition of Success	Number of Trials	Successes	Acceptable Success Rate	Pass / Fail
Packet Latency	Packet latency < 60ms.				
Successful Connections	Client requesting DNS lookup of server name is able to connect to web server.				
Connections Without Capabilities	Pseudo-IP scanning fails to detect the IP address of the web server.				
TTL Verification	Requests made within the TTL range connect to the web server; requests made outside the TTL range fail to connect.				
Correct Policy Functionality	Client that was denied a capability and had its connection blocked.				
Selectivity	All selectors in configuration files are respected.				
Policy Switching	Client succeeds at lookup before policy implementation and fails after.				
General Requirements	Pass/Fail only whether general requirements were met	N/A	N/A	N/A	

3. Results:

Overview:

In order to test the successfulness of this implementation, all of the evaluation criteria listed above were completed. The results in tabulated form (according to the provided evaluation sheet) are included below in Table II. Each section's results is then explained in more depth in the remainder of this section.

Table II: Results of Evaluating the Implementation Using the Predefined Criteria

Measurement	Definition of Success	Number of Trials	Successes	Acceptable Success Rate	Pass / Fail
Packet Latency	Packet latency < 60ms.	20	20	90%	PASS
Successful Connections	Client requesting DNS lookup of server name is able to connect to web server.	21	21	90%	PASS
Connections Without Capabilities	Pseudo-IP scanning fails to detect the IP address of the web server.	128	128	100%	PASS
TTL Verification	Requests made within the TTL range connect to the web server; requests made outside the TTL range fail to connect.	45	45	90%	PASS
Correct Policy Functionality	Client that was denied a capability and had its connection blocked.	45	45	90%	PASS
Selectivity	Capabilities can be restricted to a specific client's IP address.	15	15	90%	PASS
Policy Switching	Client succeeds at lookup before policy implementation and fails after.	20	20	90%	PASS
General Requirements	Pass/Fail only whether general requirements were met	N/A	N/A	N/A	PASS

Packet Latency

This test consisted of a comparison between the BIND9 server handling DNS requests without any capability system added, and the BIND9 server with the capability granting system. On each system, 20 trials were conducted and the mean of the trials is included in the figure below. A DNS lookup with our capability system took 37.30 ms compared to 0.63 ms on our network without capabilities. The capability system takes on average 36.67 ms, from the difference of these two results. The measurement taken before adding a capability system provided an average of the routing time, and subtracting from the average after adding the capability system provides an estimate of the time taken by the system itself.

The capability system appears in the figure to add a massive amount to the DNS lookup time, but on our isolated network, the lookup time is a small fraction of what is experienced on the internet. Compared to the DNS lookup times of around 60 ms cited in the evaluation section, our capabilities system represents an increase by a factor of 1.61, or 61%. This is an acceptable delay, by the standard that the system should cause an increase of less than a factor of two. By this standard, the capabilities system has met its goal of being lightweight in terms of time added to DNS lookups.

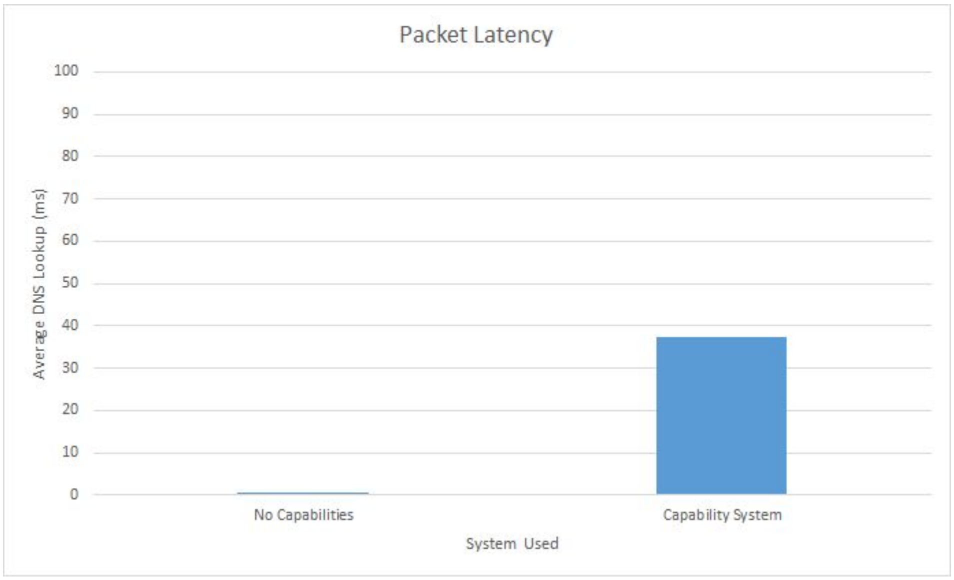

Figure 3.1: Packet Latency Evaluation

Successful Connections

For this test of the system, a script established a capability and then made a connection to the web server periodically during the TTL of capability. The TTL during this test was 15 seconds, and the system adds a grace period of 5 seconds to all TTLs before removing the relevant rules from the NAT. The connections were made every 3 seconds, and those at 0 seconds through 18 seconds were successful. This trial was performed 3 times for a total of 21 connections.

This result demonstrates that a client with capabilities was able to connect to the system as expected, with all clients being able to communicate with the web server. This test demonstrates the system's ability to process a connection attempt to the web server from a client with a valid capability. Over 21 connection attempts, all succeeded, demonstrating accessibility to authorized parties. This test is a confirmation that the system met one of the basic design requirements.

Connections without Capabilities

In this test, there was a scan conducted over the 128 addresses in the upper half of the address space of the network, the half used for capabilities. The output of the Python script that was executed to complete this test showed that all of the 128 addresses scanned were found to be unreachable.

During this scan, there was a valid capability linked to the client not conducting the scan (client 2), but the scan was not able to use this capability as its source address was that of the other client (client 1). (The “strict” option was used to restrict capabilities to the source IP address of the corresponding DNS lookup.) The lack of successful connections to the server during the scan shows that the system is resistant to scanning.

The same scenario was repeated without the “strict” option, and in this case the scan did yield a successful connection using the capability created by the other client. Without restricting capabilities to some range of IP addresses, scanning will be able to find any currently valid capabilities, but with a larger address space than was available in this implementation, and short TTL, this risk can be minimized. Also, due to the temporary nature of the capabilities, if the same address were returned to later for exploitation, it would likely not yield a connection to the web server.

TTL Verification

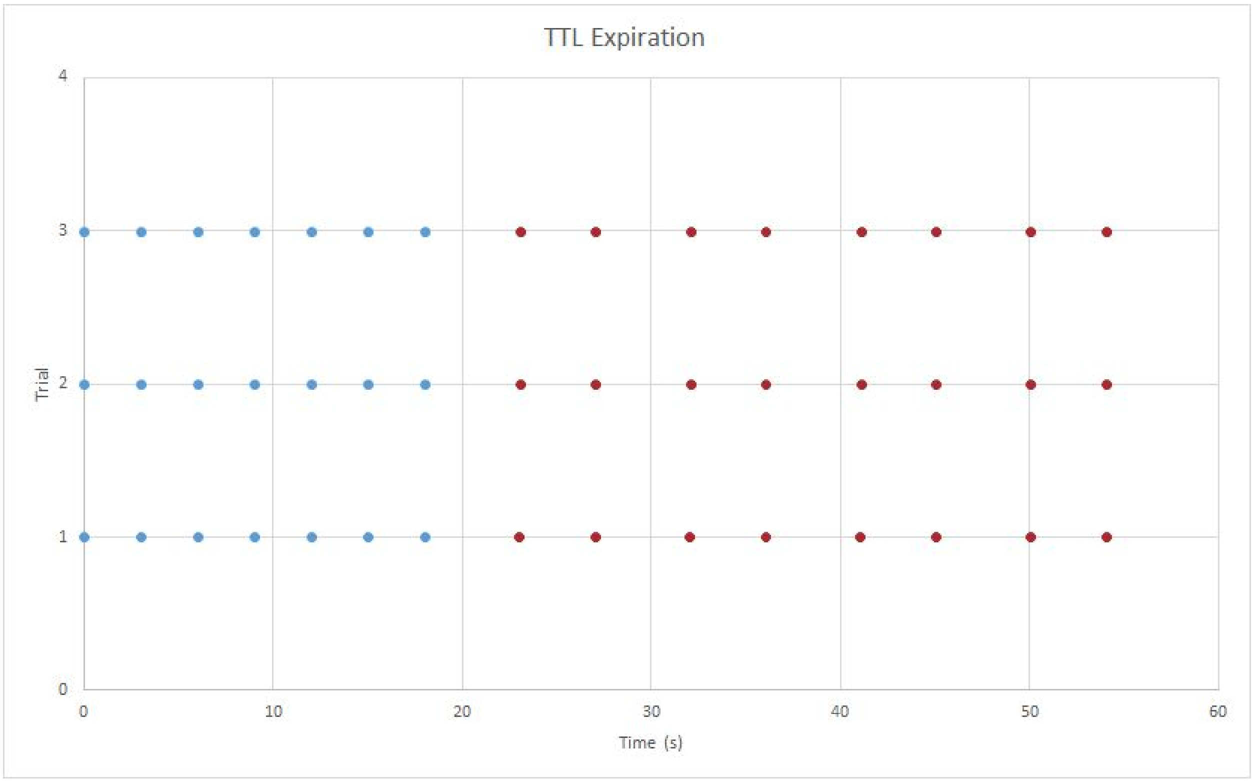

Figure 3.2: TTL Expiration - Connection Attempts to Web Server over Time

Presented above are the test trials we did with the protected server, through the router. In the chart, red dots indicate a failure to connect, while blue dots indicate a successful connection to the web server. We did tests throughout the lifetime of the capability, and then after the capability expired. With this test, we were able to correctly verify that the capabilities only lasted as long as their TTLs advertised them to last.

To test the system's adherence to the TTL values of granted capabilities, a script was run on a client to resolve the hostname of the web server, and connect to the web server every three seconds, waiting two seconds before declaring an attempt to connect unsuccessful. The TTL was 15 seconds during this trial, with expected successful results during the first 20 seconds including the grace period. The connections made were successful through 18 seconds, while those at 23 seconds and later were unsuccessful. This test demonstrates adherence to the TTL values set by the capabilities system. The results are represented in the following graph, for all three trials conducted. Each test went as expected with 7 successful connections and 8 failed connections, for a total of 45 successful trials.

Correct Policy Functionality

The correct policy functionality of the server was tested by doing the same test as the Successful Connection test, but with the IP address of the client blacklisted in the policy configuration file. We again ran 3 trials, with 15 requests per trial, for a total of 45 requests, and we were not able to connected to the protected server for any of those requests. This shows that we can correctly block hosts based off of the originating IP address. If a policy is created where the “always” field is set to “deny”, the client will not be granted a capability, and connection attempts from this client to the web server are refused.

Selectivity

Based on the previously defined evaluation criteria, the implementation of capability exclusivity was determined to be successful. In this trial, a capability was granted to one of the two client machines, with the “strict” option used to restrict the capability to only the requesting IP address. During the lifetime of this capability, both clients attempted to connect to the web server using the same test used in the Successful Connections section. The client with the capability was able to connect for each of the 7 attempts made during the TTL, and the client without the capability was not able to connect during any of the 8 attempts. This test demonstrates the system’s ability to differentiate between connection attempts to the web server based on source IP address when there are currently valid capabilities.

One weakness of this part of the implementation is its vulnerability to IP address spoofing. If the other machine to whom the capability was not distributed set the “Source” value of the IP header to be the IP address of the client who actually obtained the capability, it would be expected this client would be able to connect with their spoofed IP address.

Policy Switching

Through the evaluation metrics defined earlier, the implementation of policy switching was deemed to be successful. Figure 3.6 shows the results of the trials conducted.

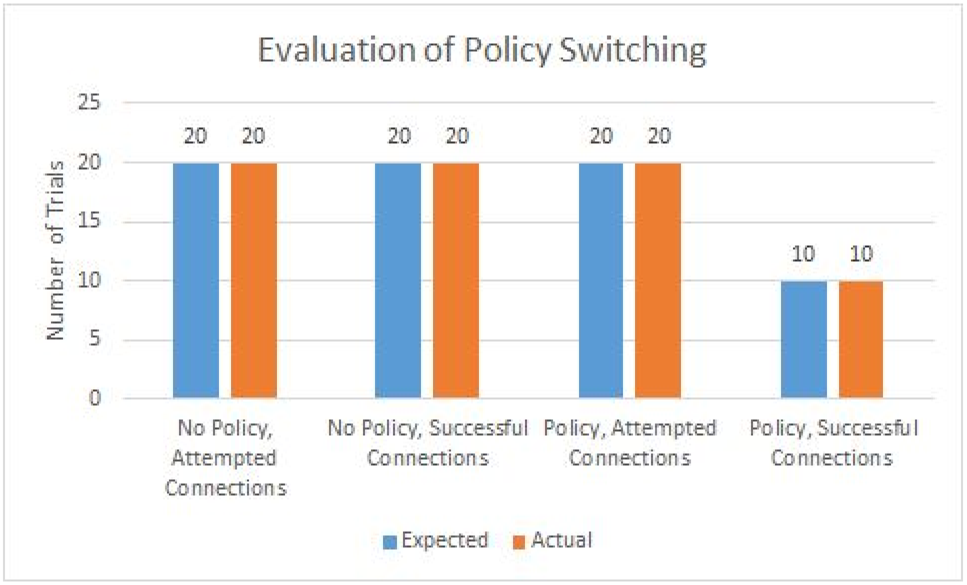

Figure 3.3: Evaluation metrics for policy switching

The trial commenced with the bare minimum capability service running, with no preconfigured policies. The two different client IP addresses were used: In this implementation, it was expected that 100% of the attempts made by clients should connect the client to the web server. This was observed to be true, as 20 connections were attempted by the clients, and all 20 of these attempts were successful.

A policy was then applied that barred the “10.4.2.1” client from accessing the network, such that only the “10.4.2.2” client was able to connect to the DNS server. Again, all both clients were expected to attempt to connect to the web server, but this time, only the “10.4.2.2” client should be capable of completing the connection to the web server. Again, the expected behavior was observed, as only the second client actually managed to connect to the network. This procedure was repeated 10 times, with changes to the policy file to restrict “10.4.2.2” and allow “10.4.2.1” for half of the cases. Because the actual measurements completely matched the expected measurements for all 20 connections (and therefore the error was less than 10%), this functionality was deemed to be successful.

General Requirements

To ensure the system was meeting all of the anticipated requirements, a variety of measurements were taken.

1. Reading and Writing to the Policy Configuration File: throughout the trials described above, the reading and writing of the configuration file failed only once due to a user typing in the file with an incorrect syntax. However, the file parser recognized the syntax error via a try-except clause and loaded a default value.
2. Updating IPTables: The system never failed to properly update iptables with new values. Both additions to and removals from the table worked without fail.
3. DNS Server Uptime: Throughout all of the trials completed, the DNS server never crashed, failed, or was overloaded. While it was impossible for us to simulate running the DNS server for immense amounts of time (weeks, months…), we believe the persistent uptime during the trials is significant to conclude that the server’s uptime is successful.
4. Web Server Uptime: Similarly to the DNS server, the web server remained active for 100% of the time during the trial. Therefore, it was concluded the uptime is successful.
5. Connections to the DNS Server: The IP address of the DNS server was static and was pre-programmed to be the default DNS server for each of the clients. As such, no client ever failed to make a connection to the DNS server; even if a capability wasn’t granted to the client, it was still able to connect to the DNS server itself.
6. Modification of Packets: The packets returned to clients were consistently processed correctly by the clients, indicating that the contents of the packets were correct. Because of this, meeting this requirement could be deemed successful.

In conclusion, the system was able to meet all of the general requirements outlined in the evaluation section.

Conclusion

Through this project, the team was able to successfully design and implement an alternative method to provide clients with the IP associated with a given hostname. This implementation shifts the load from the web server to the DNS server by using the capability functionality to drop malicious packets at the resolver. This implementation designed for this experiment was able to successfully meet all of the established design goals and criteria.

The implementation was also extended by including an easily configurable policy file that allows restrictions to be placed on specific clients or a subnet of clients. This allows the server administrator to, in the event of an attack from a particular target, quickly add policies to limit the damage that the attacker can do.

There are also many opportunities to further expand the implementation for the capability system. One possibility for expansion would be to test this network on a much larger network size. This particular implementation only used a /24 subnet, but a system like this should ideally function on much larger subnets, like a /16 subnet. The packet latency would be subject to change on a larger scale, as the current implementation uses a O(n) time to sort compare a capability request against capabilities already in the queue.

Another potential expansion of this system would be add more policy options, such as rate limiting. The policy file format was designed in such a way that new fields can easily be added and removed. It may also be worthwhile to develop a GUI that handles additions and removals to the policy configuration file, to make editing the file even easier for system administrators.

Improvements

The Abstract section and Conclusion sections were both new additions to the document.

The Design/Implementation section was largely reworked to better represent the actual system designed. A significant amount of detail was added that described many of the exact commands that would be used, instead of using vague statements about what might be done. The graphics were also updated to better show the implementation of the system. We made these changes to address peer review feedback that we needed to include more on the specific libraries being used in our functions. Furthermore, more emphasis was placed on the implementation of the policy configuration, and much of the focus was shifted away from internal and external clients. A sample of a policy file is included when the policy system is introduced to increase clarity. We also received feedback that the web server's implementation should be discussed more. To address this, we added a note to the web server's implementation section to explain its role, and the reasons we just used apache2 without making any custom code to go along with it.

In the evaluation section, all tests were elaborated on to include more specific procedures. Where use of a script was specified, the modules and methods used in the script were included, and some details including the order of these calls where relevant, and where operations like time checks were completed. Each task had a step-by-step paragraph added to explain the procedure. A specific success criterion based on the procedure was added. In the packet latency section, the success measure aspect was present before, but it was rewritten to be more clear. These changes were in response to the feedback that this section was too vague, and that we needed more specific standards to be able to state whether a test should be considered a pass or fail. This idea was present in several of the peer review feedback responses we received.

In the results section, the fake data was replaced with real information. Most sections were expanded to include some of the information from the evaluation section, as we got feedback that this section needed more of the information from the earlier sections and it was hard to understand on its own. There was also more discussion added to interpret the results, as we were told in peer review that there should be more of this. Also, we got the feedback that our charts were overall passable but lacking in format consistency, and sometimes unnecessary. Sections where the test was binary had the charts removed, and where a chart was needed, all charts were made using Excel to create a more cohesive appearance.