DNS as an Inexpensive Network Capability Provider

Devon Coleman Kyle McCormick

Chris Navarro William Van Rensselaer

ABSTRACT

There are numerous approaches for blocking unwanted Internet traffic, either once it reaches the target's network, or closer to its origin. One approach is to grant legitimate traffic access by providing it a "network capability," a piece of information that must be acquired by a client before accessing a network resource. However, using capabilities creates additional overhead in the communication process. This paper is based on [1] and discusses the implementation and testing of a lightweight capability mechanism in which IP addresses are capabilities and DNS is the capability distributer. This system also improves on the one described in [1] by showing a CAPTCHA challenge to blocked clients, giving them another chance to access the resource if the challenge is completed correctly. Although this system is not an ideal capability system, it still shows the potential of gaining key benefits while using commodity technology.

1 INTRODUCTION

A "network capability" is a piece of data that must be acquired by a client before it can access a network resource. In [1], a system is described in which the network capability is the public IP address that can be used to communicate with a network resource, and the provider of the capabilities is a DNS server. Clients that attempt to connect to a network resource, or asset, without first issuing a DNS request are sent to a honeypot server.

Under our team's approach the asset is sequestered until a capability is requested, upon which it is connected only to the requesting client. This is most effective against any scanning or probing attacks, which rely primarily on automation and therefore likely do not use DNS. Using a DNS capability distributer makes gaining a capability a simple process that can easily fit into many production networks and allows enforcement though fluxing the asset's IP address. This fluxing is done for each capability that is requested, such that a capability is only valid for the requester. This forces the DNS query to be an initial component of communication between the client and the protected asset. This scheme provides protection for the asset but shifts the possible points of attack to the system itself. However, this is a fair trade-off, given the ease of the system's implementation and security benefits provided.

This paper describes the design, evaluation, and testing of such a system with an additional improvement: clients that find themselves in the honeypot have the ability to complete a CAPTCHA to acquire a capability.

2 DESIGN

The implementation of our system contains four components: a server that performs DNS and NATing (known henceforth as the "appliance"); a Web server that is being protected by the capabilities system; a "honeypot" server, to which blocked traffic is directed; and a client that will be used to test the other three components.

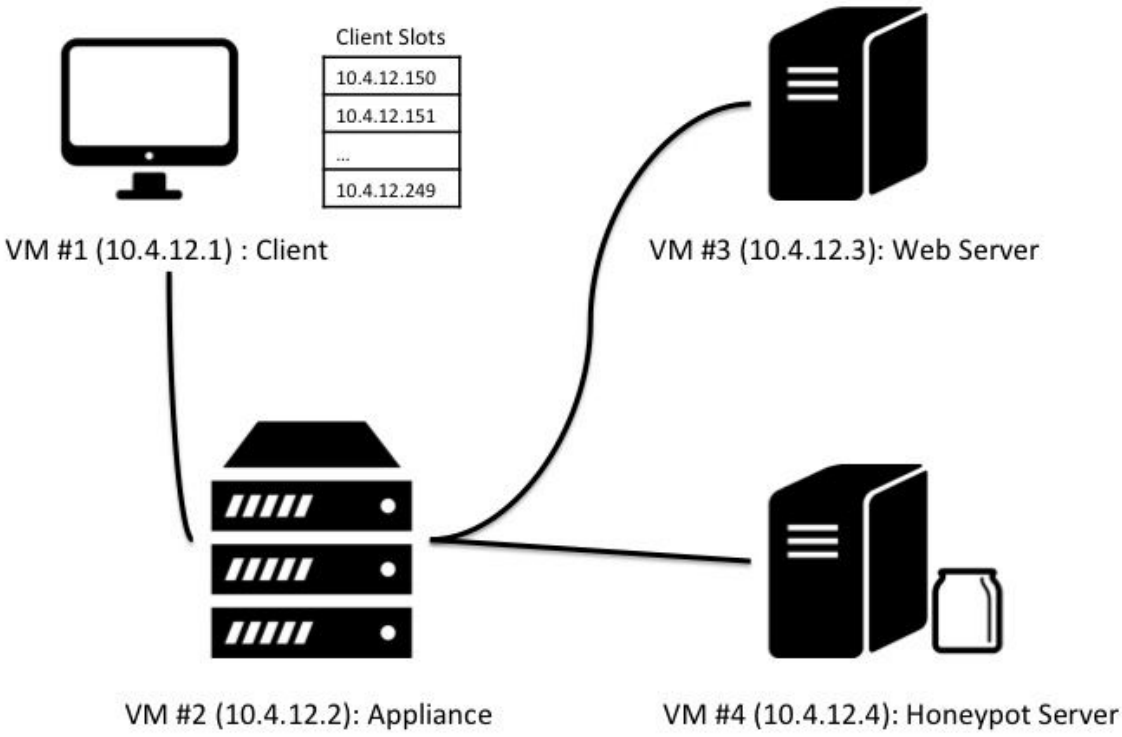

Figure 1: Legitimate Client Obtaining Capability

Appliance

The appliance serves two purposes: to distribute capabilities by handling DNS requests, and to enforce capabilities with the NAT. The DNS service is run using the Berkeley Internet Name Domain (BIND9) software package, and NAT-ing is done using Linux's built-in *iptables* utility.

Both these services are managed and coordinated by a Python script. The appliance will be run on VM #2 and will be assigned the IP address 10.4.12.2.

The appliance will expose a total of 101 virtual network interfaces. A hundred of these, in the range [10.4.12.50, 10.4.12.149], are intended for use by clients who have received an address from the DNS server, and are assigned IP addresses in this range. The remaining interface, given the address 10.4.12.255, exposes the DNS service.

The Python script uses the *netfilter_queue* library to interrupt DNS requests before they reach BIND9. Upon receipt of a request, the script chooses the next available public IP address in the range [10.4.12.50, 10.4.12.149]. It then makes a system call to *iptables* to create a rule which states that requests from this specific client to the public IP should be mapped to the Web server's private IP address. Next, the script modifies the DNS zone file (using the *dnspython* library) to map www.cap.com (the hostname of the Web server) to the public IP address with a time-to-live of 20 seconds. Finally, the script releases the packet to be handled by BIND9, which process the DNS request and responds to the client.

The Python script runs two additional threads. The first thread simply monitors NAT mappings, removing any that have existed for longer than 20 seconds. The second thread waits for TCP connections from the honeypot. When such a connection is made, the thread reads a string from the socket, and interprets this string as an ephemeral port number. Using a system call to *conntrack*, the thread looks up the client IP address that maps to the port number. Finally, the thread adds an *iptables* rule that grants the client access to the Web server using any public IP address.

It is important to note that established TCP connections are held in an IP table separate to the one used to map the public IP addresses to the Web server's private IP address. So, although mappings expire and are removed from their table after 20 seconds, established TCP connections may remain open for as long as necessary. If this were not the case, it would be impossible to download large files from the Web server, as the connection would be terminated after 20 seconds.

Honeypot

By default, clients that attempt to access the Web server via an IP address not assigned to them by the appliance are routed to this server. It serves a simple web page that provides the user with a CAPTCHA, allowing any human visitor to complete it and prove that they are in fact human. If successful, the honeypot opens a TCP connection to the appliance and sends a string representation of the client's source port number. The appliance can use this port number to

identify the IP address of the client, and make an *iptables* rule that grants the client access to the Web server. The honeypot then redirects the client to www.cap.com, which will resolve to the Web server. If the CAPTCHA fails or times out, the client will simply receive the same web page again.

The honeypot will run, on VM #4 (10.4.12.4), a Python script that implements a simple HTTP server using Python's built-in *sockets* library. It would have been preferable to use a pre-built Web server from a library, however, it was necessary to be able to determine the client's source port number. This was not possible using any of the Python Web server libraries that we considered, so it was necessary to build the server directly on TCP sockets.

Right now, the CAPTCHA is simply a static image, as the lack of Internet connection available to the honeypot made it impossible to use a legitimate service. Future implementations of this system should use a dynamic CAPTCHA service.

Web Server

The Web server, which will be hosted on VM #3 (10.4.12.3), runs on Apache. It hosts two files: a simple HTML document, and a 1.6 GB text file, which will be used in a test described in Section 3.

Test Client

VM #1 (10.4.12.1) will simulate both legitimate users of the system as well as malicious ones, such as IP scanners. The VM will perform requests to the DNS Server or the Web server. The requests will be made using built-in Linux command *curl*. Multiple clients will be simulated simultaneously by creating multiple network interface aliases and utilizing them from multiple Python threads. The aliases will be created using *ifconfig* and will be assigned IP addresses in the range [10.4.12.150, 10.4.12.249]. Clients will record time (as described in the metrics below) using the *datetime.time* type available in Python.

Component Interaction

Below is a step-by-step walkthrough of the interactions between the system components for legitimate clients who correctly acquire a capability (Figure 2), malicious clients, and legitimate users who were directed to the honeypot (Figure 3).

Valid Client with DNS Request

1) The client makes a DNS request for the Web server's hostname.
2) The DNS/NAT appliance pauses the request before it arrives at the DNS server using Python bindings for the *netfilter_queue* library.
3) The appliance selects an IP address from its set of available external addresses and creates a mapping between the IP address and the Web server's internal, private IP using *iptables*. This mapping is valid only for the client's IP address.
4) The DNS zone file is modified and the packet is released, triggering a DNS response with the newly mapped IP.
5) The client then uses the IP it obtained in step 3 to access the server.

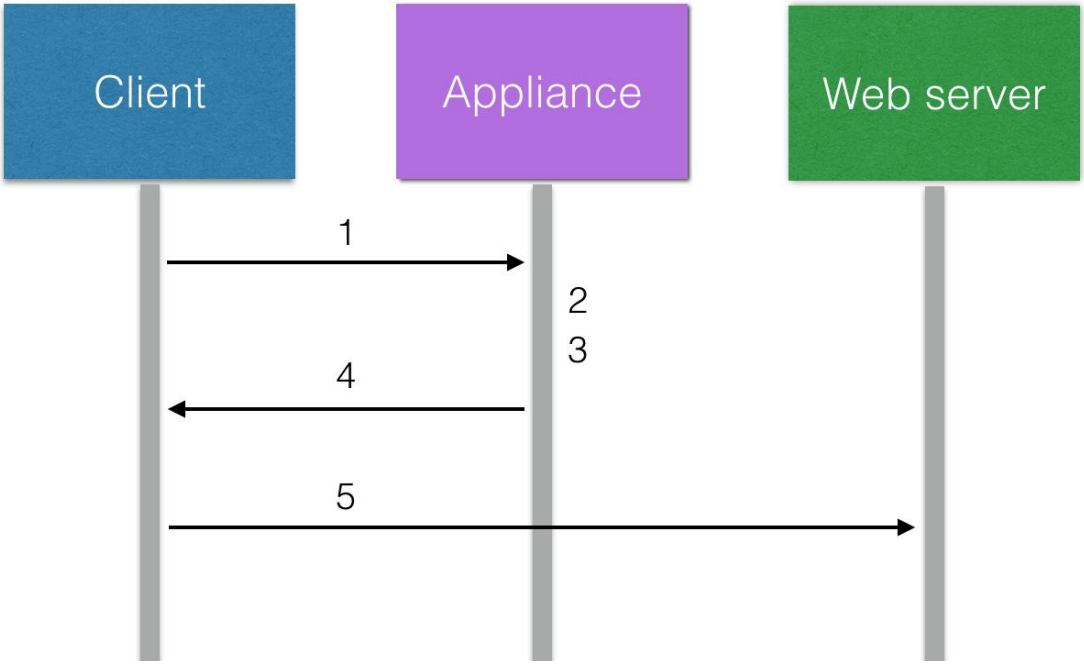

Figure 2: Legitimate Client Obtaining Capability

Valid Client without DNS Request

1) Client sends a request to an IP address that is not a valid mapping and is forwarded to the honeypot. The client correctly fills out the CAPTCHA.
2) The honeypot server detects that the CAPTCHA has been correctly solved and sends a message to the appliance containing the ephemeral port used in communication between the two.
3) The appliance uses this port to resolve the client's IP address then makes a mapping enabling that address to connect to the Web server.
4) After a short wait, the honeypot uses an HTTP redirect to cause the client to reload www.cap.com.
5) The client successfully accesses the Web server.

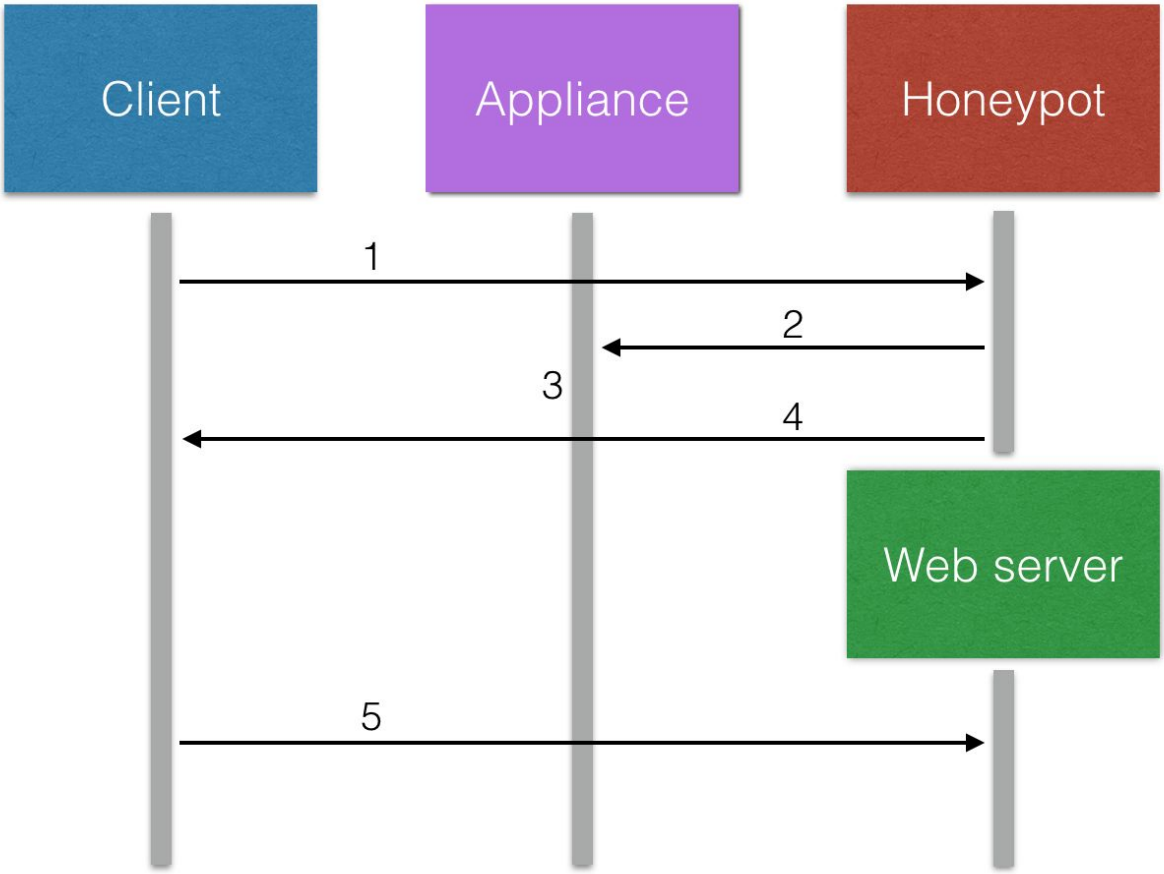

Figure 3: Legitimate Client Obtaining Capability Through Honeypot

Threat Model

Like all systems, it's important to note that ours is not not bullet-proof. It is designed to gain certain benefits of a capability system while avoiding the additional overhead and costs of implementing a system based on entirely new technology. That being said, we will describe our threat model here, and the kinds of attacks we would expect from adversaries and aim to protect against.

In our threat model we assume our main adversary to simply be a scanning bot attempting to gain access to our protected server. With this system, any bot scanning the protected network space should be redirected to the honeypot. We exclude any human adversary because it is assumed any human attempting to access the protected asset is a legitimate client.

By making the DNS server an essential part of the connection process, the adversaries' attention shifts from the Web server to the DNS server. The DNS server becomes vulnerable to a denial of

capability attack, in which it is flooded with more capability requests than it can handle. This is a potential vector by which our "automated" adversary can launch an attack. Unfortunately, when under such an attack, our DNS server will struggle to process capability requests in a timely manner, resulting in the unavailability of the asset. This vulnerability to DoS attacks, however, is shared by normal Web servers. In fact, it may be harder to overwhelm the capabilities server than it would be to overwhelm a Web server, as DNS is a lighter-weight protocol than HTTP.

Appliance Details

This section provides a description of the appliance's different functions in our implementation as most of the capability's functionality is implemented there.

Classes:

Slot

Represents an IP address slot for our mapping with timeout, client IP, mapped IP, interface, and type of mapping.

Members:

Client IP Address - String

Mapped IP Address - String

Interface - String

Type - integer: 1 if honeypot mapping, 0 if normal mapping.

Timeout- integer: The time when this mapping will expire. Set to 0 when the slot is free.

Functions:

main:

Runs all threads and begins the Netfilter_Queue packet processing.

setup:

Flushes all iptables rules and sets up all virtual interfaces. Essentially prepares the environment for appliance functionality.

iptables_cmd:

Makes a system call to the *iptables* utility using the given argument vector. Used mostly to provide locks to deal with the problems that arise when attempting to multithread the utility.

modify_dns:

This function modifies the DNS zone file to the newly mapped IP address, thus altering the DNS response to the client without mangling the packet.

on_packet_recieved:
This is the function invoked when Netfilter_Queue intercepts a packet, and is used to determine if a mapping to the server is necessary.

timeout_thread_runner
This function runs one of the three threads on the appliance. It simply tests if the timeout of the slot on top of the mapped queue been reached, and if so, removes the mapping and places the slot in the unmapped queue.

unmap_server:
This function removes the mapping to the Web server for the provided slot.

map_server:
Pulls a slot from the set of unmapped slots and maps it for the client to access the Web server.

map_honeypot:
This function maps the provided slot to the honeypot.

map_honeypot_client:
This function allows a given client that has successfully completed the honeypot CAPTCHA access to the Web server.

nat_lookup:
Uses the ephemeral port number of the connection between the appliance and the honeypot to resolve the authenticated client's IP address.

honeypot_thread_runner:
Blocks until it receives a message from the honeypot, where it will create the mapping for the requested client.

3 EVALUATION

This section outlines the evaluation methodology, metrics, and criteria we used to determine the success of our system. It is our hope that this evaluation is sufficiently detailed enough such that another networking student or professional could re-create the tests.

Metrics for Evaluating Performance

1. **DNS server response time with variable number of clients**
   We would like our setup to be such that a DNS request-response cycle takes under two-hundred milliseconds to complete. In one iteration of this test, *N* clients will make requests to the DNS server simultaneously, and their response times will be averaged and recorded. For each value of *N*, ten iterations will be performed. The values of *N* that will be tested are 1, 5, 15, 50, and 100. Although this is not a pass/fail test, we would prefer that the average response time for each value of *N* is below two-hundred milliseconds, our targeted threshold.

   Response time will be measured as follows: All clients will be spawned simultaneously on the client VM, and each will perform the same series of actions:
   a. The client will take the current time using the *datetime.time* type available in Python.
   b. The client will then initiate a DNS lookup request for the Web server's hostname to the DNS server using the dig utility included in Ubuntu.
   c. When the client receives a response from the DNS server, it takes the current time again using the same method.
   d. The client takes the difference between the two times and records it in a log file for further analysis.

Metrics for Evaluating Security

1. **Access with a capability**
   When provided a capability, a client is able to access the Web server. This is a pass/fail test; a pass means the correct webpage has been loaded, and a fail means some other scenario has occurred.

2. **Access denial without a capability**
   When a client attempts to access one of our public IP addresses that it did NOT receive as a capability from the DNS server, they are unable to access the Web server and instead are directed to the honeypot. This is also a pass/fail test; a pass means the honeypot

webpage has been loaded, while a fail means some other scenario has occurred.

3. **Access denial with an expired capability**
   When a client attempts to access the Web server with a capability that has expired (that is, the Time-To-Live field in the appliance has expired), it is directed to the honeypot. This is another pass/fail test; a pass means the honeypot webpage was loaded, while a fail means some other scenario has occurred.

4. **Access denied with another client's capability**
   Clients should be unable to access the Web server with a capability that does not belong to them. In this test, a capability will be granted to client *A*. Then, client *B* will attempt to access the Web server using *A*'s capability. This is a pass/fail test; a pass means *B* was redirected to the honeypot webpage, while a fail means some other scenario occurred.

5. **Honeypot allows server access**
   In this test, a client will attempt to access the Web server without a capability and be directed to the honeypot. There, the client will correctly complete the CAPTCHA served by the honeypot. The test will pass if the client is immediately directed to the Web server, and will fail otherwise.

6. **Persistent connections are maintained after capability expiration**
   While opening *new* connections after a capability has expired should fail, *existing* connections that were established before capability expiration should be maintained. In this test, a client will access the Web server with a valid capability and then attempt to download a large file from it. The file, whose exact size will be adjusted as necessary, will take longer than the capability's TTL to download. This test passes if the file continues to download after the capability has expired, and fails otherwise.

4 RESULTS

After implementing the system, our team ran the tests against the criteria we specified. These results, as with the criteria, are divided into two categories: *performance* and *security*. The results show that the system performed adequately and provided the intended security guarantees. Our system was, for the most part, able to successfully meet all the criteria we tested against.

Performance Testing Results

1. **DNS server response time with variable number of clients:**
   If numerous clients attempt to gain a capability through the DNS, can the DNS handle the traffic?
   When the DNS server receives a request from a client, the request is paused by the script that updates the IP mapping on the NAT, after which the request is unpaused. It is important that this interruption takes minimal time, such that that the DNS server remains responsive.

   This responsiveness was the the only performance metric evaluated, and was tested with a varying number of clients. Each client made a request simultaneously. For each set of clients the test was performed ten times and the response time of each iteration was averaged.

   Overall, our system was able to handle approximately 10 simultaneous requests before the response time rose above our intended threshold. Unfortunately, having to handle more requests than this amount causes the average response time to jump above 200 ms. At 100 simultaneous client requests the response time approached 1700 ms. The full results of the performance test are shown in Figure 4.

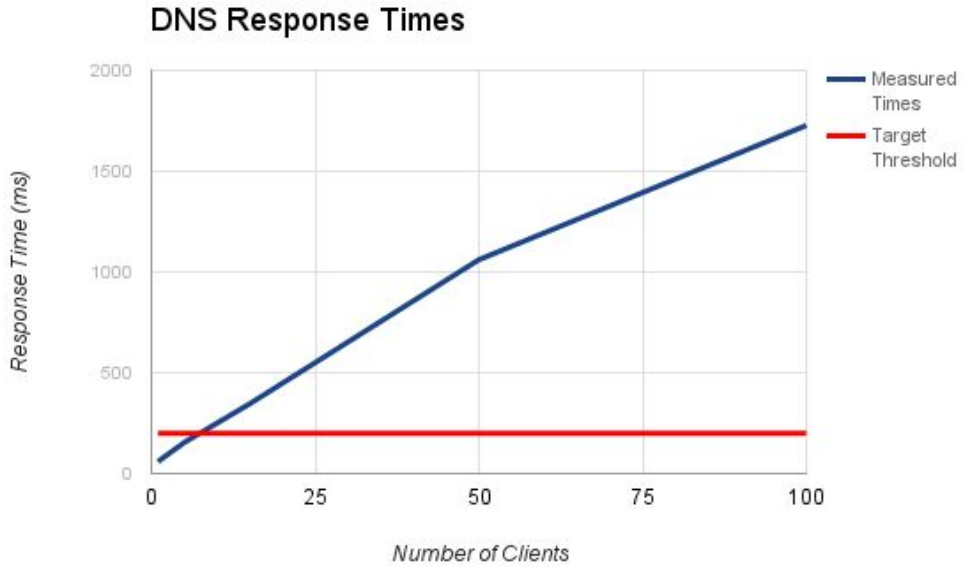

Figure 4: DNS Response Time vs. Number of Simultaneous Requests

Security Testing Results

1. **Access with a capability:**
   If a client legitimately obtains a capability, can it access the Web server?
   Clients that queried the DNS server then sent an HTTP request to the address specified in the DNS response successfully received an HTTP response from the Web server. This test was performed 10 times, the system passed each time and each client was always able to access the Web server.

2. **Access denial without a capability:**
   If a client does NOT obtain a capability, is it denied access to the Web server?
   Clients that attempted to access the Web server by sending an HTTP request to an IP address in the range [10.4.12.50, 10.4.12.149] *without* first querying the DNS server were not able to access the Web server; instead, they received an HTTP response from the honeypot. This test was performed 10 times and the system passed each time, sending each client successfully to the honeypot.

3. **Access denial with an expired capability:**
   If a client attempts to use an expired capability, is it denied access to the Web server?
   Clients that queried the DNS server, waited for the TTL of the response to expire, and then attempted to access the Web server received an HTTP response from the honeypot. This test was performed 10 times and the system passed each time, sending the clients with expired TTLs to the honeypot.

4. **Access denied with another client's capability:**
   If a client attempts to use another client's capability, is it denied access to the Web server?
   When Client A attempted to access the Web server with an IP address that was provided by the DNS server to Client B, the Client A was redirected to the Honeypot. This test was performed 10 times, the system passed each time allowing no client to gain access with another's capability.

5. **Honeypot allows server access:**
   If a client completes the honeypot's CAPTCHA, is it able to then access the Web server?
   Clients that had been routed to the honeypot (due to attempting to access the DNS server with an invalid capability) were provided a webpage containing a CAPTCHA. Clients that correctly answered the CAPTCHA received a response from the Web server, while clients that did not received a response from the honeypot. This test was performed 10

times, but divided into two parts: five where the CAPTCHA was correctly completed, and five where it was not. Each time, the system performed as expected, allowing and blocking the honeypotted clients respectively.

6. **Persistent connections and capability expiration:**

If a client's capability expires during an ongoing connection, does the connection persist?

Clients that accessed the Web server with a valid capability and then began to download a large file were able to continue downloading the file after the TTL of the capability expired. This test was performed 10 times, the system passed each time, with no clients download being interrupted.

All security tests passed against our criteria for success with zero failures. Table 1 below shows the final security testing results.

Test	**Total Number of Clients**	**Success**	**Failure**
1. Access w/ Capability	10	10	0
2. Access Denial w/o Capability	10	10	0
3. Access Denial w/ an Expired Capability	10	10	0
4. Access Denied with Wrong Client Capability	10	10	0
5.1. Honeypot Allows Server Access	5	5	0
5.2. Honeypot Prevents Server Access	5	5	0
6. Persistent Connection/Capability Expiration	10	10	0

Table 1: Security Testing Results

5 CONCLUSION

Our system and subsequent implementation of a DNS server as a light-weight capability provider was successful. All tests against our specified security evaluation passed.

One concern, however, is the performance of our overall system. When more than 10 simultaneous clients made a DNS request, the appliance was unable to process them in a timely manner, and response times jumped above our targeted threshold of 200 ms. One thing to note is that the DNS server's performance is restricted by the machine it is hosted on; our team expects that the processing power of the VM provided creates a bottleneck that would be less apparent on a system using better hardware, or a system that is load-balanced.

Overall, this system demonstrates the potential of a capability mechanism built using easily accessible and commonly available technology.

6 REFERENCES

[1] C. Shue, A. Kalafut, M. Allman, and C. Taylor. On Building Inexpensive Network Capabilities.