Using The Domain Name System to Thwart Automated Client-Based Attacks

Curtis R. Taylor and Craig A. Shue
Cyberspace Sciences and Information Intelligence Research Group
Oak Ridge National Laboratory
{taylorcr, cshue}@ornl.gov

Abstract

On the Internet, attackers can compromise systems owned by other people and then use these systems to launch attacks automatically. When attacks such as phishing or SQL injections are successful, they can have negative consequences including server downtime and the loss of sensitive information. Current methods to prevent such attacks are limited in that they are application-specific, or fail to block attackers. Phishing attempts can be stopped with email filters, but if the attacker manages to successfully bypass these filters, then the user must determine if the email is legitimate or not. Unfortunately, they often are unable to do so. Since attackers have a low success rate, they attempt to compensate for it in volume. In order to have this high throughput, attackers take shortcuts and break protocols. We use this knowledge to address these issues by implementing a system that can detect malicious activity and use it to block attacks. If the client fails to follow proper procedure, they can be classified as an attacker. Once an attacker has been discovered, they are isolated and monitored. This can be accomplished using existing software in Ubuntu Linux applications, along with our custom wrapper application. After running the system and seeing its performance on three popular Web browsers Chromium, Firefox and Internet Explorer as well as two popular email clients Thunderbird and Evolution, we found that not only is this system conceivable, it is effective and has low overhead.

1 Introduction

Attackers can compromise systems owned by other people and then use these systems to launch attacks automatically. This increases the challenge of distinguishing good and bad traffic. A researcher at McAfee recently revealed a study showing 72 network intrusions to global companies in the last 5 years, in which 68% of attacks targeted locations in the United States [2]. Since the Internet's inception, many security techniques have been introduced to prevent network attacks. One example is DNSSEC which helps to provide origin authentication and integrity assurances for the DNS messages [4]. A problem plaguing DNSSEC, and other Internet security features alike, is that it has a low adoption rate [8]. We attempt to overcome this by providing a simple, yet robust, system. In our studies, we use the DNS as an access control system by lowering time-to-live (TTL) values and periodically updating the IP addresses of all the machines on the network. In doing this, we are creating a moving target system that will help mitigate attacks. Our first method in developing this system involves manipulating the routing tables, along with IP fluxing, and the second is a Network Address Translation (NAT) based approach. Both approaches continuously update the IP addresses of servers. This means each approach is limited only by the number of addresses available in the pool of a given subnet.

2 Related Work

Attempts have been made to provide network access control by manipulating the DNS. One method allowed a server, upon receiving a DNS request from a source, to look at a blacklist and determine if the source was allowed to have access to the host the source was inquiring about. If the source was on the blacklist, an error was returned notifying the user that the host could not be located [10]. An issue with this approach is that attackers will attack a network once and not return to it rendering a blacklist pointless. Aside from the fact that a blacklist can be rendered moot, creating a blacklist is not easy. In order to add a client to a blacklist, you must tie their DNS resolver to the malicious activity, which can be challenging. Additionally, if a user is unknowingly part of a botnet and takes part in the attack, they will be blacklisted. By turning the DNS into an access control system we are helping to resolve these issues by using a method that will prevent attacks from happening.

Fast-flux service networks have been used by online scam campaigns. By making extensive use of fast-flux service networks, scammers are able to manipulate the location of malicious hosts, mitigating the effectiveness of blacklists. After comparing a large sample of scamming campaign domain IP addresses with various blacklists, Konte *et al.*discovered that as much as a third of them were not present in the blacklist [9]. Using this information, we can apply this method to our non-malicious servers. By using fast-flux as a defensive tool, we are moving our servers around so that attackers cannot access them without following protocols to locate them. Even after finding them, the location they find will not be active long enough for them to do severe damage.

One purposed solution to preventing denial of service (DoS) attacks uses network capabilities. The idea is that any node wishing to send data to another node has to obtain "permission to send" through a series of tokens given out by a request to send (RTS) server, and the tokens are to later be authenticated using validation points (VPs). Using this system, if a server was under a DoS attack, any user already connected would not see any interruption, but since all new connections must obtain an RTS, the server providing the RTS would be overwhelmed. To keep from hurting the performance of established connections, the percent of bandwidth allocated to the RTS server would be decreased [3]. For websites that are heavily trafficked, it would be preferred to not block any new users trying to connect. This weakness provoked a response by Argyraki *et al.* [5], which suggested preventing DoS attacks using their datagram approach. Unfortunately, the response is at a high-level of discussion and not an actual implementation they show would withstand an attack. They also argued that capabilities separate traffic into good and bad, and in actuality there is traffic that falls in neither category. Furthermore, by using the capabilities approach, the network is only moving the issue of a DoS attack to a denial of capabilities (DoC) attack. Similarly, an approach to preventing distributed DoS (DDoS) attacks also uses a packet marking scheme. When a router receives a packet in transmission from source to destination, it embeds a fingerprint in every packet. The advantage of using this scheme is evident when considering the traditional method of trying to trace a packet's path back to a source. Normally a large number of packets are needed before being able to figure out the entire path, but with a fingerprint scheme, this process is faster. The downside to this method is that it would require at least half of the routers in the path to follow the scheme and does not provide much flexibility to the user [13]. Although these are generally automated attacks, our research does not attempt to solve the problem of DoS or DDoS directly, and our system may still be susceptible to this type of attack.

With the increasing number of phishing scams, browsers have started implementing phishing toolbars to help prevent users from accessing known phishing pages. Abu-Nimeh *et al.*, discovered that these toolsbars are easily deceived [1]. They created a scenario involving the use of a rogue access point (AP) and showed that if a user connected through them, the attacker could use DNS poisoning to connect the user to any IP address the attacker chose. To verify this address, the user's browser tried a DNS lookup to verify the address, but since it went through a rogue AP, it appeared legitimate. All of the toolbars tested failed to recognize this attack. Even when not connected to a rouge AP, a DNS server is still vulnerable to poisoning. These types of network attacks are extremely dangerous to users due to the stealthiness of the attack. Often the target of attack for DNS poisoning is the recursive server. An attacker will submit a query causing the recursive sever to send out a request to resolve the domain. When this happens, the attacker has a fairly good chance of guessing the 16-bit ID field that the recursive server is expecting the SOA to return with, giving the attacker the ability to poison the server. Generally, with minor exceptions, the query is copied exactly the way the initiator's packet was sent, which can cause unnecessary lookups (e.g., "`goOgLE.COm`" vs "`google.com`"). A very efficient and simple solution to this is to convert all queries into lowercase and then use 0x20 encoding so that all entries in cache will be the same, reducing the chance of poisoning [6]. Although there is not much that can be done for connecting to a rouge AP, DNS poisoning is a concern for our approach. With short TTLs, there will inevitably be more DNS lookups that take place. For this reason, a simple and efficient 0x20 encoding method might be preferred to use alongside our system.

3 Methodology

To show that our system works, we created a network in a laboratory environment and ran various tests. We now describe our experimental designs and results.

3.1 Tools Utilized

All machines were connected via a network switch and running the Ubuntu 11.04 operating system with the exception of the client machine that was also running a virtual machine hosting Windows XP - SP2 and Windows 7 Professional. The Web browsers used were Mozilla Firefox 5, Google Chromium 12.0.742.91 (Ubuntu)/10.0.648.134 (Windows), and Internet Explorer 8 (Win 7)/6 (Win XP). The mail transfer agent (MTA) used was Postfix 2.8.2, and the mail applications tested were Thunderbird 3.1.11 (Ubuntu)/ 5.0 (Windows) and Evolution 2.32.2 (Ubuntu). Apache 2.2.17 was also installed as the Web Server, and BIND 9.7.3 was installed to handle the DNS. Finally, Wireshark 1.4.6 was installed for capturing network traffic.

3.2 Network and System Configuration using Routes

In our test network for the first approach, we used four machines running Ubuntu 11.04 as shown in Table 1. The first three machines were connected using an Ethernet switch. The first machine, designated as our router, mail server, and the authoritative DNS server, had two network cards and the fourth machine, which we designated as our client, was connected through the router's second network card. The second machine was configured as a Web server while the third was configured as a honeypot.

Table 1: Computer list for Experiment 1

ID	Role	IP Address	Route(s)	Gateway
1	Router and DNS server	192.168.1.1 10.0.0.1 Dynamic: 10.0.0.2 10.0.0.3	192.168.1.0/24 10.1.1.1/32 10.0.0.0/8 10.0.0.2/32 10.0.0.3/32 10.8.4.9/32 10.7.3.6/32	Direct (eth0) Direct (eth1) 10.1.1.1 (eth1) Direct (eth1) Direct (eth1) Direct (eth1) Direct (eth1)
2	Web server	Dynamic: 10.8.4.9 10.7.3.6	0.0.0.0/0	10.0.0.1
3	Honeypot	10.1.1.1	0.0.0.0/0	10.0.0.1
4	Client	192.168.1.100	0.0.0.0/0	192.168.1.1

We now describe each of these machines in greater detail:

- **Machine 1**: This machine acted as our router, mail server, and the DNS server. We installed Postfix 2.8.2 to handle mail transfers, and BIND 9.7.3 to create a local DNS server with authority for `example.com`. The zone file was configured to have a 10 second refresh, retry, expire, and negative cache. In the zone file, we had `A` records for Machine 2 and for the mail server (Machine 1's dynamic address), which were modified as needed. A query for either machine name returned the most recently assigned IP address of the host. The router had routes for the mail server's aliases, each alias of Machine 2, a route for Machine 3, and a route for 10.0.0.0/8, using Machine 3 as the gateway, all through Interface 1. Finally, it had a route for 192.168.1.0/24 for Machine 4 through Interface 0.

- **Machine 2**: The Web server, with Apache 2.2.17 installed, is designed to be the "protected server" in the network. In this machine's Apache `httpd.conf` file, two changes were made. One change disabled ETag and Last-Modified headers so every access to Apache appeared as a new page request and was not already stored in cache. The second change allowed the execution of CGI scripts. The second machine used two IP aliases on its network interface. Since it used two aliases, the machine could accept connections from the previous IP address, which was necessary since the DNS server may have provided the old IP address immediately prior to the rotation, meaning that the IP would need to remain valid until the TTL on that record expired, which the usage of a prior alias supports. This machine had a single entry in its routing table for the gateway, Machine 1.

- **Machine 3**: Machine 3 was designed to be a honeypot computer where all traffic was sent by default. This machine intercepted all expired IP addresses from Machine 2 as well as any other IPs that are unused in the 10.0.0.0/8 prefix.

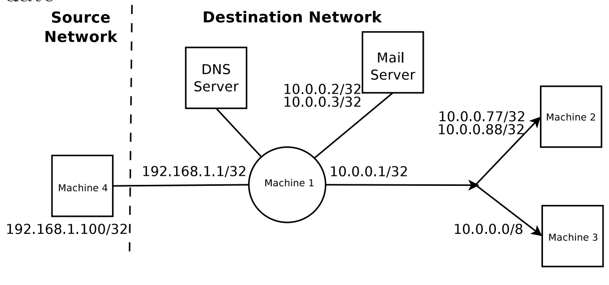
- **Machine 4**: Our client machine ran Linux natively, but also used VirtualBox to host two Windows virtual machines. The guests were Windows XP service pack 2 and Windows 7 Professional. The machine had variants of the top three Web browsers installed on the host and two guests: Mozilla Firefox 5 (on the host and both guests), Google Chromium (version 12.0.742.91 on the host, version 10.0.648.134 on the guests), and Internet Explorer (version 8 on the Windows 7 guest, version 6 on the Windows XP guest). Together these browsers comprise approximately 94% of Web traffic [12]. Chromium had two features for speeding up a user's network browsing experience: network and URL prediction. For timing reasons, we disabled these features. This machine also had two mail applications installed on Ubuntu, Thunderbird 3.1.11 and Evolution 2.32.2, and one installed on both versions of Windows, Thunderbird 5.0. The client used static IPs for each of the guests and the host OS. Each were in the 192.168.1.0/24 subnet. Each of these OSes had a static route for its gateway, Machine 1, and each were configured with Machine 1 as their DNS server.

3.3 Experiment 1

The system launches from a program on Machine 1, and begins by traversing the zone file. For each `A` record, a random IP address is generated, within the subnet of the network, to replace the current IP. As each IP is changed, the old address becomes the second alias on the respective machine, and the new address becomes the newest IP alias. This is done remotely from the router. After all records have been updated, BIND is issued a reload to reflect the new information in the zone file. As soon at BIND has successfully reloaded, any client that attempts to access a server that has just had its `A` record updated will receive the IP address just placed in the zone file.

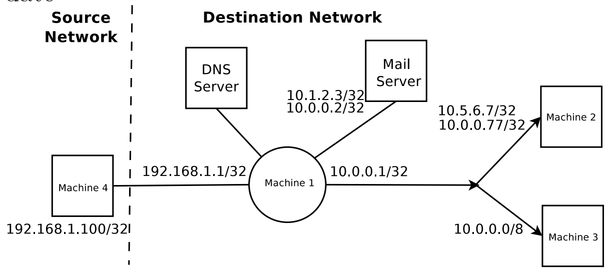
To give a more lucid and in depth description, we will give an example of how the update process works with all four of our configured machines. Figure 1 shows an example network.

Figure 1: Configuration for Experiment 1 Before Update



In this network, there are two `A` records, that of the mail server and Machine 2. Traversing the zone file, each record is found and is assigned a random IP address, in this case 10.1.2.3 and 10.5.6.7. The zone file is then changed to reflect this. The aliases are then added to each machine as the first alias replacing the old first IP, for example in Machine 2 10.0.0.77 and moving it to the second alias replacing 10.0.0.88. With the introduction of new IP addresses to the network, the routing tables must be updated or the 10.0.0.0/8 route of the honeypot will be taken for this new address. Therefore, each newly generated IP has a route added with the gateway of Machine 1, and the old route is deleted. Since the old routes no longer exist in Machine 1 and it falls in the 10.0.0.0/8 range, anyone who tries to access an old address will be seeing our honeypot. At this point, BIND is given a reload command. The system then waits until the TTLs of the new aliases are approaching expiration and need replacing. Figure 2 shows how the original example network would appear after this test run.

Figure 2: Configuration for Experiment 1 After Update



3.4 Experiment 2

The introduction of `iptables` is what defines experiment 2 since it allows us to implement an efficient method of NAT. For this experiment, the major adjustment takes place in Machine 1. The only other change was to Machine 2 and 3 which now only have static addresses and no longer have aliases on their machines. Everything will be handled at the router. Therefore, we will only explain the new configuration for Machine 1.

- **Machine 1**: This machine acts as our router, mail server, and the DNS server. We installed Postfix 2.8.2 to handle mail transfers, and BIND 9.7.3 to create a local DNS server with authority for `example.com`. The zone file was configured to have a 10 second refresh, retry, expire, and negative cache. In the zone file, we have `A` records for Machine 2 and for the mail server, which can be modified as needed. A query for either machine name returns the most recently assigned IP address of the host. The router has routes statically configured to Machine 2, Machine 3, and a route to 192.168.1.0/24 for Machine 4 through Interface 0. Finally, there are six entries in the NAT table. The first two entries are that of the mail server redirecting traffic from its alias to its static IP. The next two entires map Machine 2's aliases to its static IP, and the fifth entry allows access to the DNS server by allowing traffic to 10.0.0.1/32 to be redirected to itself. The last entry maps 10.0.0.0/8 to Machine 3, the honeypot.

Table 2 shows how routes are configured for this experiment. These routes are never altered after load time.

Table 2: Routing table for example network

ID	Role	IP Address	Route(s)	Gateway
1	Router and DNS server	192.168.1.1 10.0.0.1	192.168.1.0/24 10.0.0.10/32 10.1.1.1/32	Direct (eth0) Direct (eth1) Direct (eth1)
2	Web server	10.0.0.10	0.0.0.0/0	10.0.0.1
3	Honeypot	10.1.1.1	0.0.0.0/0	10.0.0.1
4	Client	192.168.1.100	0.0.0.0/0	192.168.1.1

Table 3 shows how the NAT table will appear on Machine 1. Here, the two aliases for the mail server and the two aliases for Machine 2 are the first rules in the PREROUTING table in `iptables` to be addressed. If one of these rules fit, the traffic will be forwarded to the corresponding redirect address. In order to avoid all traffic accessing the DNS server being redirected to the honeypot, we include a rule for 10.0.0.1 to be redirected to itself. Since no rule in the NAT table fits 192.168.1.0/24, the routing table will control the flow of this data.

Table 3: NAT table on Machine 1 Before Update

Destination	Redirect to
10.0.0.2/32	10.0.0.1 (eth1)
10.0.0.3/32	10.0.0.1 (eth1)
10.0.0.77/32	10.0.0.10 (eth1)
10.0.0.88/32	10.0.0.10 (eth1)
10.0.0.1/32	10.0.0.1 (eth1)
10.0.0.0/8	10.1.1.1 (eth1)

Similar to that of the first approach, we start by traversing the zone file. When an `A` record is found, we generate a new address to replace the old. As soon this address is replaced in the zone file, it is added to the NAT table using `iptables` to insert a rule. By using insert, we add the new rule to the top of the PREROUTING table so it will be priority over other rules, and by inserting this rule before restarting BIND, the new IP address is actually accessible before the zone file is reloaded to reflect the changes. After all changes are made to the zone file, BIND is restarted. Lastly, the old aliases are removed from the NAT table. Using the same random addresses as experiment 1, Table 4 reflects the changes to the NAT table.

Table 4: NAT table on Machine 1 After Update

Destination	Redirect to
10.1.2.3/32	10.0.0.1 (eth1)
10.0.0.2/32	10.0.0.1 (eth1)
10.5.6.7/32	10.0.0.10 (eth1)
10.0.0.77/32	10.0.0.10 (eth1)
10.0.0.1/32	10.0.0.1 (eth1)
10.0.0.0/8	10.1.1.1 (eth1)

3.5 Web and Mail Tests

We performed our Web browser and mail client tests in the same way for the experiments.

In order to test how the browsers performed, we navigated to Machine 2's Web server via its domain name. Following this, we issued HTTP requests by either refreshing every 10 seconds or navigating to another link provided on the page but did so consistently. If the browser failed to perform as expected with our 10 second TTL, we continued issuing HTTP requests until normal activity resumed, which is described in the results section.

The approach used to test the mail clients was similar to that of the Web browsers. We configured each client using POP/SMTP for incoming and outgoing mail and attempted to retrieve new mail using each application's "Get Mail" function every 10 seconds. The mail server was configured to have the same incoming and outgoing domain so that trying to receive mail would test the same network features as trying to send mail.

4 Results

We now explain in detail the results from experiment 1, the approach using routing tables to manipulate data flow, and experiment 2, where we used NAT.

One valid concern was whether or not ARP entries would take precedence over routing and NAT tables resulting in data being sent to the wrong location. Considering a situation where a client was actively accessing an address associated to Machine 2 that was about to expire. After the address is retired, subsequent traffic should be sent to the honeypot, but the router will have an entry the ARP table associating this old IP to the MAC address of Machine 2, and it will be labeled reachable, but as it turns out, this does not disrupt traffic.

Across both experiments, several other details were uncovered relating both to the operating systems and browsers. First, Ubuntu, and many other Linux operating systems, do not have pre-installed caching capabilities. An application such as the name service cache daemon (`nscd`) could be installed to provide caching capabilities, but we chose to use the default Ubuntu configuration without the cache. Windows XP and 7 both have a built in the DNS cache. This cache is viewable via command line using `ipconfig /displaydns` and will display all the current entries in the DNS cache along with corresponding values including that of the TTL. In both versions of Windows, the OS cache respects TTL values, allowing them to timeout properly. This means if you use a program such as `ping` that uses the OS's cache, it will perform a DNS query after the TTL is expired and will never pin it.

On the other hand, browsers have their own internal cache and behave differently. When testing each browser, we saw different types of behavior when given different variables. For instance, each browser reacted differently when the honeypot would drop packets instead of accept them or when the user navigated to links listed on the Web page versus refreshing the page. As a result, before giving the results on how the experiments carried out, we will first try explain the Web browser's behavior. There were two main reasons for the varying behavior found. The first reason was whether the user was actually refreshing the Web site or merely navigating the site through links listed on the page. Obviously, the latter is the more common scenario. The other aspect causing variation was whether or not our honeypot was silently dropping packets or accepting them. Table 5 shows how this behavior varies across each OS and browser.

Table 5: Dropping Packets

Operating System(s)	Browser(s)	Behavior - (Refreshing/ Navigating Internally)
Windows XP Windows 7	Chrome Firefox	Query every 10 seconds/ Timeout after 21 seconds
Ubuntu	Chrome Firefox	Query every 10 seconds/ Timeout after 190 seconds
Windows XP	IE6	After 21 seconds, sends query/ Timeout after 21 seconds
Windows 7	IE8	After 21 seconds, connection fails/ Timeout after 21 seconds

If the honeypot were dropping packets, Chrome and FF would respect our TTL if the client manually refreshed. With both of these browsers accepting such low TTL values and not pinning them, they are also more susceptible to DNS rebinding, but as discovered by Jackson *et al.* [7], DNS pinning is ineffective in modern browsers because of vulnerabilities introduced by plug-ins. On the other hand, if the client were browsing inside the Web page (i.e., navigating links listed on the page), these two browsers would behave differently depending on if the honeypot was dropping packets. If packets were being dropped, both browsers would timeout after about 21 seconds on Windows and after 190 seconds on Ubuntu. IE6 behaved differently from Crome and Firefox when the honeypot silently dropped packets. If the user requested a refresh on IE6, the browser would try the address cached for about 21 seconds then would issue a DNS request and the page would update accordingly. Additionally, on IE6 if the user navigated internally, after about 21 seconds the page would fail to connect and display a 404 error to the user. Interestingly, if the user then requested a refresh, IE6 would issue a DNS request and update. This differs from IE8, which never issued a DNS request (until the 30 minute TTL expired) if the honeypot was dropping packets. It was also found that after IE8 fails to connect, if you use the "Diagnose Connection Problems" feature, behind the scenes IE8 will send a DNS request, get the new IP address, and then successfully use the new address for a GET HTTP. IE8 then proceeds to tell you that "Troubleshooting could't identify the problem" and continues to leave the failed IP in cache. Although it would have been possible to change the DNS cache length on IE for possibly better results, we did not take this approach [11]. As for configuring the honeypot to accept packets, we found normal DNS pinning for each browser as shown in Table 6.

Table 6: Dropping Packets

Browser(s)	Operating System(s)	Length of DNS Pin
Chrome	Windows XP Windows 7 Ubuntu	1 minute
Firefox	Windows XP Windows 7 Ubuntu	3.5 minutes
IE6	Windows XP	30 minutes
IE8	Windows 7	30 minutes

Mail clients did not appear to be affected by the honeypot's decision regarding dropping packets. Evolution was found to never implement DNS pinning. Conversely, Thunderbird pinned for 3.5 minutes on all three OSes. Although the applications did handle packets being dropped well, the amount of time needed before the application's connection would timeout while trying to connect was different between Ubuntu and Windows. On Ubuntu, it took about 2.66 minutes to timeout and about 21 seconds on Windows.

Another possible concern for our approach is the amount of time it takes BIND to restart on each reload. We found that by using `rndc reload example.com`, reloading a new zone file took about 13.6ms. The results took effect immediately as the restart finished. In the case that someone accessed the original domain during the reload, BIND would simply use the IP address stored in its cache.

4.1 Experiment 1 Results

This approach works but has several issues. Existing connections are broken when a route is dropped. Also, any time a host's IP address was changed, the address had to be added on the host as an alias and updated in the router and the DNS server. We did this using SSH, and the time it took to update an address was about 323ms.

Additionally, this means that even though the 10.0.0.0/8 route caught all traffic not directly associated to a host, the honeypot still needed to have the address the client was trying to access in order for traffic to be routed to it, and in order to catch all traffic from expired addresses, the honeypot would have to have many aliases.

4.2 Experiment 2 Results

In experiment 2 we chose to implement NAT using `iptables`, and as a result, we gained several benefits. These benefits include more control over what is going through the network with the ability to regulate certain protocols and controlling the ports they use, but the main reason for choosing this approach was because it fixed the dilemma that involved breaking connections. Fortunately, if there is an active connection (e.g., uploading an image) on a host's IP address when the IP address expires and moves to the honeypot, the active connection is kept open until the transfer finishes. This is handled in the Linux kernel. When an address in the `iptables` NAT table is accessed, the connection is loaded into the kernel's NAT table. With the `iptables` NAT table, we hold the information for all new connections, and in the kernel's NAT table, we maintain a list of existing connections. Once the IP address a client is using expires, and the transfer is finished, the connection is closed and cleared from the kernel's NAT table. Hereafter, any connections to this address will be associated with the honeypot.

Upon further exploration of NAT tables, it was also discovered that they are considered before any routes. From this, we see that NAT can be used to control the initial flow of packets. Therefore by using NAT as a firewall, we can stop certain protocols and ports at the router and avoid unnecessary network traffic by choosing not to send that data through the network to the host. Because of this, if the data makes it past the NAT tables, we can more confidently pass the data to the routing table to correctly route the data where it needs to go. This allows for routes to remain static after load time.

This approach also removes any need for aliases on machines, which took about 323 ms. This process is replaced by adding entries to the NAT table, resulting in a much more efficient process. On average, it takes about 3.6ms to add or remove an entry from the NAT table.

5 Conclusion

We seek to turn the DNS in an access control system using one of our two mentioned methods. Our first method provides an easy way to filter network traffic based on solely using the built in `route` program on Linux. While this approach uses only routes, it is limited for several reasons. The host machines must know all of its aliases in order for routes to work properly. Every machine must be updated with all new address, and updating every machine consistently and efficiently can be costly depending on the size of the network. Active connections will also be lost when a change in addresses occur. The second approach provides solutions to these difficulties by making the routes static to the host machines and only cycling entries in the NAT table. The latter method also addresses having to give aliases to every machine, and perhaps the biggest issue this method solves is not dropping an active connection with the IP being used is cycled to the honeypot.

The introduction of this approach allows for organizations to seamlessly create a moving target system to help protect their machines from malicious activity. Incorporating our methods means that attackers will have to follow proper rules in order to access data on local host machines. To see how well our system worked with a Web server, we tested three different Web browsers and two of these browsers were also tested on three different operating systems. We found that Chromium and Firefox would allow at least a 4 minute TTL without issues, and Internet Explorer would allow for a minimum of a 30 minutes TTL.

Acknowledgments

Notice: This submission was sponsored by a contractor of the United States Government under contract DE-AC05-00OR22725 with the United States Department of Energy. The United States Government retains, and the publisher, by accepting this submission for publication, acknowledges that the United States Government retains, a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this submission, or allow others to do so, for United States Government purposes.

References

[1] S. Abu-Nimeh and S. Nair. Bypassing security toolbars and phishing filters via DNS poisoning. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1 –6, 30 2008-dec. 4 2008.

[2] Dmitri Alperovitch. Revealed: Operation shady rat. `http://blogs.mcafee.com/mcafee-labs/revealed-operation-shady-rat`, August 2011.

[3] Tom Anderson, Timothy Roscoe, and David Wetherall. Preventing internet denial-of-service with capabilities. *SIGCOMM Computer Communication Review*, 34:39–44, January 2004.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), March 2005.

[5] Katerina Argyraki and David Cheriton. Network Capabilities: The Good, the Bad and the Ugly. In *Fourth Workshop on Hot Topics in Networks*, November 2005.

[6] David Dagon, Manos Antonakakis, Paul Vixie, Tatuya Jinmei, and Wenke Lee. Increased DNS forgery resistance through 0x20-bit encoding: security via leet queries. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 211–222, New York, NY, USA, 2008. ACM.

[7] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from dns rebinding attacks. In *Computer and Communications Security*, pages 421–431, 2007.

[8] A. J. Kalafut, C. A. Shue, and M. Gupta. Touring DNS open houses for trends and configurations. *IEEE/ACM Transactions on Networking*, PP(99):1, 2011.

[9] Maria Konte, Nick Feamster, and Jaeyeon Jung. Fast flux service networks: Dynamics and roles in hosting online scams. Technical Report GT-CS-08-07, Georgia Institute of Technology and Intel Research, 2008.

[10] S.A. Strentzsch and L.T. Donzis. Method and apparatus for providing network access control using a domain name system, July 2001. US Patent 6,256,671.

[11] Microsoft Support. How internet explorer uses the cache for dns host entries. `http://support.microsoft.com/kb/263558`, 2011.

[12] W3Schools. Web statistics and trends. `http://www.w3schools.com/browsers/browsers_stats.asp`, June 2011.

[13] Abraham Yaar, Adrian Perrig, and Dawn Song. Pi: A path identification mechanism to defend against DDoS attacks. In *In IEEE Symposium on Security and Privacy*, pages 93–107, 2003.