Collaborative Client-Side DNS Cache Poisoning Attack

Fatemah Alharbi*†, Jie Chang‡, Yuchen Zhou§, Feng Qian¶, Zhiyun Qian*, and Nael Abu-Ghazaleh*

* Computer Science Department, University of California Riverside {falha08@,zhiyunq@cs.,nael@cs.}ucr.edu

† Taibah University, Yanbu, Saudi Arabia fmhharbi@taibahu.edu.sa

‡ LinkSure Network, China changjie@wifi.com

§ Information Assurance Department Northeastern University zhou.yuc@husky.neu.edu

¶ Computer Science Department University of Minnesota – Twin City fengqian@umn.edu

***Abstract*— DNS poisoning attacks inject malicious entries into the DNS resolution system, allowing an attacker to redirect clients to malicious servers. These attacks typically target a DNS resolver allowing attackers to poison a DNS entry for all machines that use the compromised resolver. However, recent defenses can effectively protect resolvers rendering classical DNS poisoning attacks ineffective. In this paper, we present a new class of DNS poisoning attacks targeting the client-side DNS cache. The attack initiates DNS poisoning on the client cache, which is used in all main stream operating systems to improve DNS performance, circumventing defenses targeting resolvers. Our attack allows an off-path attacker to collaborate with a piece of an unprivileged malware to poison the OS-wide DNS cache on a client machine. We developed the attack on Windows, Mac OS, and Ubuntu Linux. Interestingly, the behaviors of the three operating systems are distinct and the vulnerabilities require different strategies to exploit. We also generalize the attack to work even when the client is behind a Network Address Translation (NAT) router. Our results show that we can reliably inject malicious DNS mappings, with on average, an order of tens of seconds. Finally, we propose a defense against this type of poisoning attacks.**

I. Introduction

The Domain Name System (DNS) is an essential component for the Internet: it provides resolution primarily of Fully Qualified Domain Names (FQDNs) (*i.e.,* human-readable domain names such as `foobar.com`) to their corresponding Internet Protocol (IP) addresses. DNS resolution information is maintained by a hierarchical and distributed set of name servers in order to support scalability and to enable distributed management at each individual organization. DNS queries from clients are serviced using a set of resolvers that can walk the DNS hierarchy to reach an authoritative name server that provides the answer to the DNS query. To improve performance, resolvers and end hosts heavily use caching, exploiting locality to avoid unnecessary and slow queries that consist of several round-trips as a resolver walks the DNS hierarchy.

The security of DNS is critical to the security of the Internet: if an attacker can manipulate the mapping, she can redirect connections to cause users to access a malicious server, to facilitate Man-in-the-Middle (MitM) attacks, or to cause denial of service (DoS). One of the most serious attack classes against DNS is the cache poisoning attack, where an attacker attempts to inject malicious DNS mappings to the cache of a DNS resolver [1]–[4]. To protect against cache poisoning attacks on DNS resolver caches, Source UDP Port Randomization (SPR) [2], [5] was introduced and is currently widely deployed. In this defense, a DNS query from a resolver uses a random source UDP port when forwarding a DNS query. An off-path attacker must guess this random port number in order to successfully spoof a reply to the same port (otherwise, the reply will not be accepted), in the time window before the true response is received. Although it does not close the vulnerability completely, SPR substantially reduces the chances of effective cache poisoning attacks. Even though fundamentally secure DNS protocols such as *DNSSEC* [6] have been proposed, it has been difficult to get traction with respect to real-world deployment, and most websites continue to run insecure versions of DNS.

In this paper, we introduce a new and dangerous DNS poisoning attack targeting the end user devices. Most operating systems on client devices use DNS caches that retain DNS responses and share them across applications including browsers. We show that these caches can be compromised via a DNS cache poisoning attack oftentimes in a couple of seconds for Windows and a few minutes for Ubuntu Linux and Mac OS. Specifically, the attack is initiated by an *unprivileged* malicious program (*e.g.,* a malware or a malicious JavaScript) who simply asks for DNS resolution for a domain it is attempting to poison. The malicious program coordinates with an off-path attacker (*i.e.,* an attacker anywhere on the Internet) that responds to the DNS request attempting to poison the cache entry and succeeding with high probability. To succeed in the attack, a malicious response with a matching TXID has to arrive before the real response. This is challenging task as there is really only an attack window equivalent of a round-trip time between the client and resolver. To make matters worse, once the authentic DNS response is cached, one may need to wait for the entry to timeout before being able to launch the next round of attack on the same domain. However,

we discover that *specific OS implementations* and *real-world TTL/network latency* make our proposed attack highly feasible. We also consider the scenario where the victim is behind a Network Address Translation (NAT) router. Through analysis of NAT implementation on commercial routers, we show reliable strategies to launch the attack even through a NAT router.

Client devices are typically not considered to be part of the DNS hierarchy and therefore have not been considered by defenses against DNS cache poisoning. Thus, defenses against resolver cache poisoning attacks including SPR [2], [5] and 0x20 [7] do not protect against this new attack. Even new proposals such as DNSSEC which rely on cryptography to completely close cache poisioning [8] operate at the resolver level but leave the network behind the resolver unprotected. As a result, the attack represents a new and dangerous vulnerability that threatens most computing devices. The attack also expands our understanding of the threat surface of DNS cache poisoning attacks when designing mitigations within DNS.

The paper also contributes a lightweight client-side defense strategy to mitigate this vulnerability. In particular, although the attack significantly reduces the entropy by removing the uncertainty regarding the source UDP port number, it still relies on guessing the transaction ID (TXID) field, which has a range of 2^{16}. The attack is successful because today's DNS clients simply discard illegal DNS responses that do not match the port and TXID of a pending request. In contrast, our defense first detects a suspected attack when it encounters DNS replies with the wrong TXID. Once an attack is detected, the client can respond in a number of ways to mitigate the attack while maintaining the ability of the client to continue to resolve DNS requests. The defense requires only a client-side patch.

Disclosure. We reported the attack to Apple, Microsoft, and Ubuntu. Apple intends on issuing a security advisory along with mitigation to the vulnerability. We also understand that Microsoft and Ubuntu are considering mitigation strategies.

II. Background–Cache Poisoning Attacks

DNS cache poisoning is a dangerous class of attacks that has been the focus of studies in the past [9]–[11]. In 2007, Amit Klein introduced sophisticated cache poisoning attacks against BIND 9 DNS resolvers [4] and Windows DNS servers [12]. At that time, the attack's entropy was totally based on the TXID, and the implementation of the randomized algorithm facilitated the attacks. In 2008, Dan Kaminsky presented another significant attack [2] against DNS resolvers which also depends on TXIDs for authentication. The attack assumes both the UDP source and destination ports are fixed as 53. Indeed, Paul Vixie already reported this vulnerability in 1995 [10], and in response, Bernstein proposed a challenge-response defense to substantially use ephemeral ports randomization in order to expand the entropy of the correct response packet [13], [14]. However, this solution was not practically supported until Kaminsky's attack [15]–[17].

More recently (starting in 2011), several new cache poisoning attacks against resolvers were proposed which have varying degrees of assumptions of the attack requirements and the network [18]–[23]. For example, Herzberg and Shulman [18] propose an attack that exploits packet fragmentation of UDP packets of long DNS responses to spoof the second fragment of a DNS response (only the first fragment includes the TXID). The same authors [21] propose a poisoning attack that exploits delegation of DNS resolution where intermediary network devices perform recursive lookups on behalf of the resolvers. In contrast, they also show attack principles (but do not demonstrate the attack) [19], [20] that can poison the cache of a DNS resolver located behind a NAT.

Because attacks that undermine DNS resolution are extremely powerful, several defenses were proposed to address DNS cache poisoning attacks. The preponderance of these defenses targets improving DNS resolver security and therefore are not effective against our attack. The defenses can be classified into three categories: challenge-response defenses, cryptographic defenses, and using alternative architectures. Challenge-response defenses rely on the idea of increasing the entropy of DNS request/response, such as UDP source port randomization [2], [5], 0x20 encoding [7], random selection of authoritative name servers [15], [17], and adding random prefixes to domain names [24]. Challenge-response defenses are vulnerable to MitM adversaries who can intercept the DNS communication. To protect against this type of eavesdropping attack, cryptographic defenses were proposed which include primarily DNSSEC [6] that is based on digital signatures for authentication; this solution in principle closes all cache poisoning attacks since the validity of the response is no longer only a function of the contents of the packet. Despite the fact that DNSSEC is effective, the deployment is very slow. For example, a recent study [25] discovered that 0.67%, 0.91%, and 0.91% of .com, .net, and .org Top Level Domains (TLDs) are signed. Recently, Klien et al. did an Internet-scale measurement study on the vulnerability of DNS resolvers and discovered that 92% of resolvers are vulnerable to at least one type of poisoning attack [23]. Unless DNSSEC is extended to cover end clients, which introduces a substantial key distribution problem and is likely to infringe on usability, it does not prevent our attack. Recent defenses such as DNS over HTTPS [26], DNS over TLS [27], and DNScrypt [28], have been proposed primarily to preserve the privacy of DNS traffic. These proposals can also have the side effect of hardening DNS traffic against injection attacks. Although there are standardization efforts behind these proposals, they are not yet widely deployed.

A third defense alternative considers rethinking DNS implementation radically, resulting in different security properties. For example, Schomp et al. [29] propose a radical change to the DNS ecosystem by eliminating shared resolvers entirely to have clients perform the recursive resolutions directly. However, since our attack targets the endpoints, it should still be effective against this architecture.

III. Attack Fundamentals and Threat Model

Modern OSes have built-in DNS caches. These caches are shared OS-wide, meaning that if an application populates an entry in the cache, this entry will be used by any other application that requires resolution for the same name. Similar to records cached at the DNS resolvers, an OS-wide DNS cache record is stored along with a Time-To-Live (TTL) value which is set by the domain authoritative nameserver to determine the lifetime of the record in the cache. The purpose of having such a cache is to improve the performance of DNS resolution as it is on the critical path of accessing Internet resources, especially for applications that have many short-lived connections.

We surveyed a number of modern operating systems, including macOS Sierra version 10.12, several versions of Microsoft Windows (Microsoft Windows 7 Professional Edition, Microsoft Windows 8.1, and Microsoft Windows 10), as well as several Linux distributions. By default, the OS-wide cache is enabled in all versions of Windows, Mac OS, and in Ubuntu 17.04 and later. It is checked by DNS APIs such as `getaddrinfo()`: only if the record is not found in the cache, does the DNS cache service perform a DNS request on behalf of the application. Thus, this client-side cache acts as the *de facto* first level of resolution.

The OS-wide DNS cache is stored in memory. We measured the size of the cache starting with an empty cache, warming the cache with a number of domain names (say x), and then resolving these names again while timing to see if the entry is being resolved from the cache. We keep increasing x until we start observing misses, which identifies the size of the cache. We found the cache size to be 2050, 5076, and 4094 entries in Windows, Mac OS, and Ubuntu Linux respectively. Furthermore, we found that the OS-wide DNS cache in all operating systems stores all types of records (A, AAAA, CNAME, PTR, RRSEG ...etc.) from only the *answer section* of DNS responses. Thus, Kaminsky's attack [2] relying on malicious records from the *additional section* does not apply to OS-wide DNS caches.

Threat Model. We consider four entities below. (1) The victim client machine and its OS-wide DNS cache. (2) A legitimate resolver which acts as a DNS server for the client machine. The client may connect to this resolver via a NAT device which has its own DNS cache. (3) The on-device malware, which is unprivileged and cannot tamper with other applications directly (we will instantiate this later in §IV). This malware could be a malicious JavaScript running in the browser after a victim browses a malicious/compromised website, or a malicious application downloaded to the user's phone. (4) The off-path attacker, who is capable of spoofing the IP address of the legitimate resolver. The malware and off-path attacker collaborate to poison the OS-wide DNS cache with a malicious mapping for a target domain name.

Note that the IP spoofing capability of the off-path attacker is commonly available in networks unless ingress filtering [30] is implemented [31]. A significant number of Internet Service Providers (ISPs) and networks do not implement ingress filtering and therefore an attacker connected to such a network can directly spoof IP addresses [32]–[34]. This threat model matches the threat model used in recent papers (e.g., off-path TCP injection attacks [35], [36]).

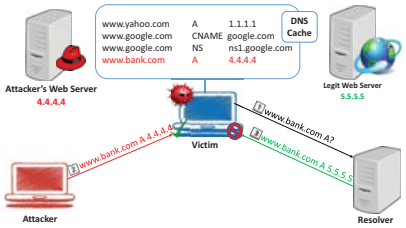


Fig. 1: High Level Attack Overview

IV. Attack Construction and Analysis

We first overview the attack at a high level and introduce possible attack scenarios. We then explain how to overcome a number of challenges needed to implement the attack, leading to a complete end-to-end attack under realistic conditions.

A. Attack Overview

The basic attack is overviewed in Fig. 1. A malicious user-level program requests a DNS resolution for the target DNS domain (in this example, *www.bank.com*). The goal of the attacker is to poison the cached DNS entry such that it resolves to the IP address of an attacker server redirecting user connections targeting *www.bank.com* to go to the attacker's web server. Once the DNS request is sent, the attacker attempts to respond with fake responses. DNS resolvers accept the first correct response: a response with both a matching port number so that it is received correctly, and a matching TXID field; if a correct response from the attacker is received before the response from the DNS authoritative name server, the client simply accepts this response and caches it in its OS-wide DNS cache. The attacker's response uses a large TTL to ensure that the poisoned value remains in the cache. Any future connections from this machine to *www.bank.com* will redirect to the attacker's server.

B. Attack Scenarios

The attack requires a malicious user level program to execute on the victim machine. We consider two main scenarios for launching the attack.

• Public/shared machines. such machines are commonly found in many places including universities, libraries, hotels, and stores. Any user who can log in to the machine can run a malicious program and collaborate with an off-path attacker to conduct the attack, poisoning the DNS cache, and leaving the machine to be used by victims. For all tested operating systems, we find that the OS-wide DNS cache is in fact *shared across multiple users*. This means that a malicious user (*e.g.,* guest) capable of poisoning the OS-wide DNS cache can cause a different user (admin or guest) to also use the poisoned cache. Furthermore, for Windows, any user (including guest)

can clear the cache directly without requiring admin privilege, so that the malware can clear legitimate entries to make room for poisoned entries. We confirmed, after obtaining permission from the system administrators to conduct an experiment then clearing the cache, that shared public machines in four large universities are vulnerable to the attack.

• Malware. Applications downloaded from an App store, or malicious JavaScript on a website that is malicious or compromised, can also be used to launch the attack. In the public machine scenario, the attacker may need physical access to the machine. In this attack scenario, the victim unknowingly downloads a malicious application that launches the attack, without requiring physical access to the machine.

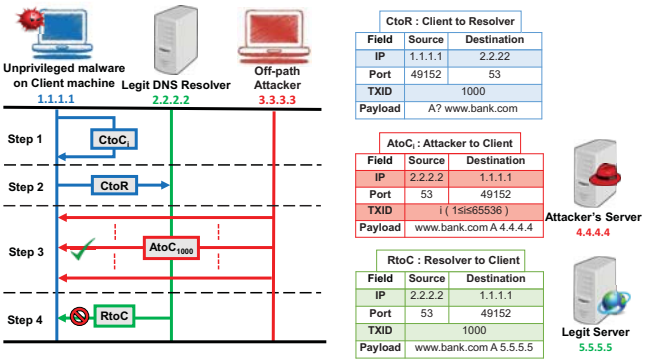


Fig. 2: Design of client-side DNS cache poisoning attack

C. Challenges and Detailed Attack Procedure

Source Port Reservation. In order to send spoofed responses, the off-path attacker must first obtain the DNS request's source IP address, source UDP port, destination IP address, and destination UDP port. IP addresses of the victim and resolver can be obtained easily using the unprivileged malware on the victim through standard OS interfaces, and the well-known destination UDP port for DNS requests is 53. The final challenge is to identify the source UDP port. As stated in RFC 6056 [16], the 16-bit dynamic port range of UDP is 49152 through 65536 which is typically used in Windows and Mac OS operating systems. However, we found out that in Ubuntu Linux, the ephemeral port range is 32768 through 60999.

There exist many port selection algorithms [16]. Many are proposed specifically to defeat port predictions which means the port allocated each time for a new socket will appear to be random; however, the work in [20], [22] show that these algorithms are not efficient. Without documentation, it is unclear which algorithm is used in Windows or Mac OS. Nevertheless, after testing the source UDP port number selection of the DNS services in Windows and Mac OS, we find that they appear to be unpredictable.

A basic building block of our attack is the ability to predict or infer the source UDP port of a DNS request. Based on our measurements, we discover that surprisingly all operating systems we tested are permissive in terms of the number of simultaneously open sockets they allow to any program. This allows an application (*e.g.,* malware) to reserve all local port numbers but one so that the system DNS service will be forced to pick the one and only available port. Specifically, in Windows, any unprivileged application by default can open as many UDP sockets as desired and bind to a selected ephemeral port number. In Mac OS, there is a limit of the system resources consumption (which is 10240 file descriptors by default) but can be raised to a higher number (*e.g.,* 100,000) without root privileges [37]. Likewise, Ubuntu Linux has a default limit of 4096 file descriptors for each process which also can be raised to meet the attack requirements [38]. Even without raising the per-process limit, we can simply create a single application to fork multiple child processes (*e.g.,* 2 and 6 processes in Mac OS and Ubuntu Linux respectively) to be able to reserve the required number of ports. We have verified that Android have similar behaviors to Mac OS and Ubuntu Linux.

Cache Poisoning. After the port reservation is done, the cache poisoning attack is started. The unprivileged malware on the client machine contacts the off-path attacker machine to coordinate the attack. We assume there is only one unoccupied UDP port (*e.g.,* port 49152) and the target domain name is *www.bank.com*.

The steps of the attack are illustrated in Figure 2. First, the malware on the client machine, at address 1.1.1.1, reserves all UDP ports except 49152. Second, the malware triggers a DNS query, denoted by $CtoR$, for the target domain name *www.bank.com* to the legitimate DNS resolver at address 2.2.2.2 with the source UDP port of 49152. The client OS randomly selects a TXID (say 1000). Third, the attacker repeatedly sends spoofed responses, denoted by $AtoC_1$, $AtoC_2$, ..., $AtoC_{65536}$, each with a different TXID field. If one of these responses contains the correct TXID (which is $AtoC_{1000}$), the cache can be poisoned to store a malicious IP address for *www.bank.com*.

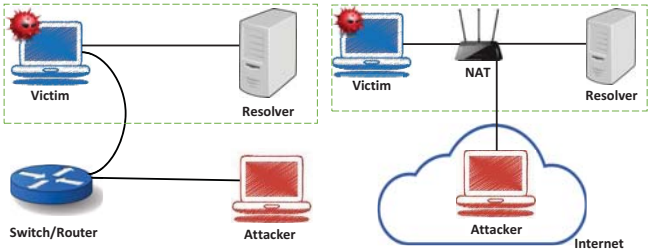
Since TXID is a 16-bit field, a brute-force attack is possible even though the number of guesses seems large, especially given the attack may need to repeat over many trials (*i.e.,* by simply issuing `getaddrinfo()` calls). Finally, the resolver responds to the DNS query issued by the malware and sends an authentic response, denoted by $RtoC$. However, the response is ignored by the DNS system service since there is no longer a pending query. Otherwise, the authentic IP address will be cached and the attacker will repeat the attack starting from the second step. The steps are the same as if the client is behind NAT except that the attacker tries to poison the response of the NAT instead of the resolver.

V. Tailored Attack Strategies and Analysis

In this section, we consider OS-specific attack strategies depending on whether the client is behind a NAT router.

A. Attacks without considering NAT

In this scenario, we assume the attacks are against networks without a NAT device on the route between the off-path attacker and the victim client (Fig. 3-a). This setting is



(a) Client without NAT (b) Client Behind NAT

Fig. 3: Attack Network Topologies

common; many networks do not use NAT. Moreover, if the attacker and victim are in the same wired or wireless LAN, then a NAT will not be traversed.

To succeed in the attack, a malicious response with a matching TXID has to arrive before the real response. On paper, this may be a challenging task as there is really only an attack window equivalent of a round-trip time between the client and resolver. Even with a high bandwidth, the attack has a fairly low probability of success. Assuming an RTT of 5msec (client and resolver are close), and an attack bandwidth of 10,000 spoofed responses per second, only 50 packets will be received before the authentic response, leading to a success probability of $\frac{50}{65536} = 0.076\%$. To make matters worse, once the authentic DNS response is cached, one may need to wait for the entry to timeout before being able to launch the next round of attack on the same domain. However, we discover that *specific OS implementations* and *real-world TTL/network latency* make our proposed attack highly feasible. We next detail three OSes: Windows, Mac OS, and Ubuntu Linux.

1) Windows: We observed an interesting behavior when testing the attack on Windows. In particular, we find that the DNS system service (*e.g.,* `getaddrinfo()`) retransmits the request as soon as it receives a (spoofed) response with an incorrect TXID, and it simply aborts after five failed retransmissions. We reverse engineered the DNS system service binary (`dnsapi.dll`) and found that it indeed has a simple loop with `select()` for up to five times. Thus, if the legitimate response is preceded by five spoofed responses, it will not be accepted (since the request resets, and the new request has a different TXID). This makes the attack much easier as the authentic response can hardly be cached when attack traffic is present. Meanwhile, although the attacker only gets five chances to guess the TXIDs before each invocation of `getaddrinfo()`, the attacker can always call `getaddrinfo()` multiple times until the attack succeeds.

2) Mac OS: For Mac OS, we find that unlike Window's 5 response limit behavior, Mac's DNS system service continues to accept DNS responses until receiving a response with a matching TXID. If none is found before the timeout, it will simply retransmit and continue to wait for the correct response to arrive. This means that the attack window is only a single round-trip time before the legitimate response is received. This makes the attack window somewhat limited: if we can send 10K spoofed messages per second, and $RTT = 5msec$, then

TABLE I: TTL for Alexa top 10 global websites

Rank	Website	TTL (seconds)	Rank	Website	TTL (seconds)
1	Google.com	60	6	Reddit.com	30
2	Youtube.com	60	7	Yahoo.com	60
3	Facebook.com	60	8	Google.co.in	60
4	Baidu.com	300	9	Qq.com	20
5	Wikipedia.org	600	10	Taobao.com	180

TABLE II: The average RTT (in milliseconds) for Alexa top 500 global websites from different vantage points

DNS Server	VP1	VP2	VP3	VP4	Google Cloud	EC2
Google DNS	61	141	62	57	36	50
Quad9	104	191	109	103	77	70
OpenDNS	70	156	68	57	60	55
Norton	34	121	38	65	293	35
Comodo	164	185	130	80	82	94
Level3	58	136	77	71	58	58

the number of chances is 50. Also, if the current attempt fails, we have to wait for the cached authentic response to timeout before we can retry (recall that this is not the case for Windows whose cache can be cleared even by a non-administrator user, *e.g.,* guest). While this slows down the attack, we found the TTL values are typically short. Table I shows the TTLs of the top 10 global websites based on Alexa [39]. We find that 58%, 27%, and 19% of the Alexa top 500 global websites have a TTL value less than or equal to 60sec, 30sec, and 20sec, respectively. Importantly, since at the start of every attempt, the value has expired in all the caches, this gives us *an RTT window of the full resolution through the DNS system*, traversing several resolvers, which can be in the tens if not hundreds of msecs. Thus, the attacker gets a larger number of guesses before the authentic response is received. To confirm the larger RTT time, we tested the RTTs from 6 different vantage points (VPs) including EC2 and Google cloud servers and 4 large universities to 6 public DNS servers. The dataset is the top 500 global websites based on Alexa [39]. We used `nslookup` to forcibly send a DNS query to open DNS servers (*e.g.,* 8.8.8.8 for Google public DNS server) regardless of its caching status. As shown in Table II, the RTTs are indeed relatively high.

3) Ubuntu Linux: The first release of Ubuntu Linux that supports OS-wide DNS caching is Ubuntu 17.04 that uses `systemd-resolved` as the default system DNS service [40]. We find that the behavior is almost identical to Mac OS. The main difference appears to be that when all UDP ports are reserved on Ubuntu Linux, we find that DNS queries can still be sent to the resolver using random source TCP ports. To overcome this problem, we use the same port reservation technique in §IV-C to reserve *all* TCP ports and force the DNS service to use the one and only available UDP port for all outgoing queries. For completeness, we also measured the behavior of `dnsmasq`, a popular DNS/DHCP software [41], for other Linux distributions which behave very much the same way as Mac OS and Ubuntu. In other words, our attack is also effective on any Linux-based system running this DNS API.

To further improve the attack time for MacOS and Linux, we can try to accelerate flushing of the DNS entry from the cache, for example, by filling the cache with new requests. In addition, we developed a strategy to interfere with the resolver's ability to respond to the DNS requests, which provides

a larger time window for the attacker to operate. Specifically, the attacker can launch a DoS attack that floods the external resolver with spoofed DNS requests for domain names different than the one the attacker targets (*e.g.,* www.bank1.com, www.bank2.com, ... etc) to fill its socket buffer. The idea is that since we can predict the source UDP port of the DNS request, all spoofed DNS requests will target the same exact socket buffer to which the real DNS request will also go. If the legitimate DNS request is received while the socket buffer is full, it is dropped and no response is sent back. For example, we configured our own DNS server which runs Ubuntu Linux 14.04 LTS, and we measured the per-socket buffer for the OS and found that the default per-socket buffer size is 17KB which can hold only $\simeq$ 300 DNS packets.

B. Client behind NAT Attacks

As shown in Fig. 3-b, a NAT allows clients on a private network to connect to the Internet by remapping their private addresses to its own IP address and using port numbers to keep track of the mapping of internal connections to external ones. With respect to our attack, since the attacker does not generally know the external port assigned to the DNS query, NAT increases the entropy. Fortunately, we found that under several popular settings, we can derive the external port allowing the attack to proceed. We tested three NAT devices (Linksys WRT3200ACM, Netgear WNDR4500 v3, and Netgear WGR614 v9). We found the NAT's port translation behaviors for DNS sessions depends on how DNS is configured on the client and on the NAT:

1. **Both Client and NAT Use DHCP.** By default, DHCP configures both the client and NAT to use the local DNS server. In this scenario, the NAT translates the source UDP port of the client to a random port each time the client contacts the local DNS server.
2. **Client Manually Configures DNS.** When a client changes the DNS settings to an alternative DNS server (*e.g.,* 8.8.8.8 for Google public DNS), the NAT preserves the source port of UDP sessions to that DNS server. Using an open resolver is becoming increasingly common: a recent study shows that 12.70% of a sample of size $\simeq$ 735 million machines around the world use Google Public DNS server [42]. Moreover, there are other Open DNS servers that are popular (*e.g.,* Quad9, OpenDNS, Norton, Comodo, and Level3).
3. **NAT Manually Configures DNS.** If the client uses DHCP but NAT uses a manually configured DNS server, the NAT assigns a fixed source port to all DNS sessions to that DNS server.
4. **Both Client and NAT Manually Configure DNS.** In this case, similar to Case 2, the NAT preserves the client's source UDP port.

Our attack can be easily carried out in all above scenarios except Case 1: in Cases 2, 3, and 4, the external source UDP port is known to the attacker, and the attack can be easily modified to attack a client behind NAT. For Case 1, we can bootstrap our attack by using principles proposed by Herzberg et al. [19], [20] to reserve the source port. In particular, this attack creates a large number of connections to attempt to reserve all the external source UDP ports of the NAT router. If only one port is left, the NAT router is forced to use it and the attacker no longer has to guess the port number.

C. Performance Analysis

For a single spoofed reply, the probability of making a correct guess for the TXID is 1 out of 2^{16} (2^{16} is the TXID field range). The probability of failure of the attack in response to a single invocation of `getaddrinfo()` is determined as follows: $P(failure) = \frac{2^{16}-z}{2^{16}}$, where z is the number of guesses up to a maximum of 2^{16}. For Windows, z is 5 since the `getaddrinfo()` fails after 5 tries, leading to a success rate of 0.0076% for each try. The overall success of the attack over x invocations of `getaddrinfo()` is: $P(X \leq x) = 1 - P(failure)^x$. The average number of trials before a success is determined by the geometric distribution and is $\frac{1}{1-P(failure)}$. For Windows, this comes out to be a little over 13000 tries. Since Windows does not rate limit `getaddrinfo()`, each attempt can take as short as 2msec, which means that the average time to succeed will be as short as $2msec * 13K = 26$ seconds.

For Mac and Linux, z is determined by the bandwidth of the off-path attacker to the client and the RTT of the DNS resolution (which determines when the legitimate response is received). For example, pessimistically assuming a low RTT of 5msec, and an attacker bandwidth of 10K spoofed messages per second, z is 50 packets, leading to a success rate of approximately 0.076% (about 10 times that of Windows), leading to the average number of trials before the success of just over 1300. However, since each retry has to wait for the value to expire from the DNS cache (*e.g.,* 30 seconds), the time until success can be long (close to 11 hours for this example). We note that this is highly pessimistic since the uncached RTT is much higher than 5msec. For instance, using the average value in Table II which 92msec, z is 920 packets, leading to a success rate of approximately 1.4% and an average number of trials before the success of 70. In this case, the time until success is dropped to 35 minutes. The analysis does not take into account network effects (e.g., dropped packets due to overrunning buffers) which we found to have significant (even dominating) effects in some scenarios.

VI. Evaluation

In this section, we experimental ly assess the effectiveness of the end-to-end attack in realistic settings for all our attack scenarios.

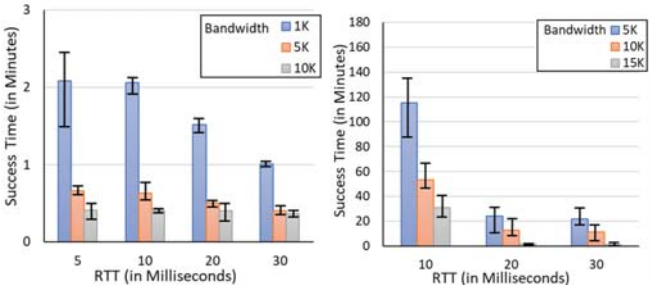
Client Platform. We conduct our experiments on client machines running Microsoft Windows 7, 8.1, and 10, MacOS Sierra, and Ubuntu 17.04 which all support an OS-wide DNS cache. We note that most other Linux distributions, including Ubuntu versions prior to 17.04 do not enable an OS-wide cache by default, although DNS cache implementations could be installed for example by enabling `dnsmasq` [41].

Network Configuration. We use two different network topologies mirroring those shown in Fig.3-a (without NAT) and Fig. 3-b (with NAT). For the NAT-based attacks, we configure an internal DNS resolver, using the BIND name server software (BIND9) on Ubuntu 14.04 LTS; (which we denote by `U14`). The NAT acts as port-preserving, attempting to allocate the same outside port as the inside port, (case 2 as described in §V). We also verified the attack on fixed-port NATs (case 3) and found it to be easier since there is no need to reserve the ports on the client machine.

The network bandwidth between all nodes is 1Gbps. In a real attack environment, there may not be such a high bandwidth and we therefore intentionally limit the attacker's throughput using the Linux Traffic Control utility `tc` [43]. TC allows us to configure the Linux packet scheduler to simulate lower bandwidth connections and also control the effective RTT to the attacker. In addition, we emphasize that since the off-path attacker knows which source UDP port the DNS request will use, she can start flooding the client/NAT with spoofed responses even before the client initiates the DNS request. With this strategy implemented, the attacker has a full round-trip time window to try and guess the correct TXID.

Experimental Details. We show the measurement results of the attack against all operating systems. Each data point is generated by 50 repeated experiments. Given the large dispersion in the time to succeed (due to the geometric distribution of the number of tries until success of the attack), we estimated that bounding the confidence interval of the mean requires many thousand experiments in several of our scenarios. This is not feasible since some experiments take on the order of hours. Thus, instead of using the mean, we use the median of 50 experiments for each point since the median is more robust to outliers. We also calculate the 95% confidence intervals of the medians and show those on the figures. Note that confidence intervals for medians are not symmetric around the median. We use the formula in [44] to calculate the *rank order* of the upper and lower bounds instead of their actual values. For one representative case, we show the histogram of the time to succeed for individual experiments to provide insight into the distribution of the time to success of the attack for the different operating systems.

A. Results of attacks without considering NAT



(a) Windows (b) Ubuntu Linux

Fig. 4: Median time to an attack success without NAT

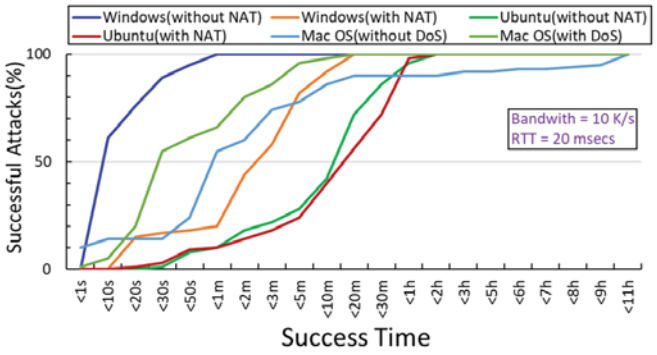


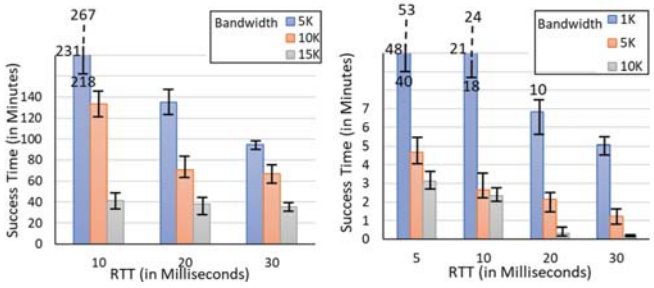
Fig. 5: Success Time Statistics

1) Windows: Fig. 4-a shows the measurement results of the attack against Windows. The time to succeed is small, one the order of 10s of seconds, in most configurations as shown in Fig. 5, even though the number of trials to succeed is typically in the tens of thousands. The fast success time is possible because `getaddrinfo()` can be invoked extremely quickly— most of the time `getaddrinfo()` returns in less than 2 milliseconds after 5 spoofed responses are received. In practice, however, we do observe that rarely ($< 0.1\%$ of the trials) the authentic response arrives within the first 5 responses due to network jitter, especially as RTT gets larger. Furthermore, the results show that the likelihood of a successful attack depends significantly on the two factors (RTT and bandwidth): the more resources the attacker has the faster the attack can succeed.

2) Ubuntu Linux: Fig. 4-b shows the attack against Ubuntu. Since the malware does not have access to flush the DNS cache (as in the Windows attacks), we consider an attack on a website with a TTL of 30 seconds (which is in the common range experimentally measured in §V-A). Thus, for each failed trial, the attacker waits for 30 seconds before the next trial. As shown in the figure, the attack time-to-success also depends on RTT and bandwidth.

Note that in this scenario (as well as MacOS), every trial waits until the TTL expires before it is initiated. Thus, the response to the query will miss the caches and likely go through the full resolution step, or hit a cache upstream, resulting in a large delay until the authentic response is received. We measured this delay to be on average 92msec (See Table II). Thus, for the attack on Ubuntu and Mac OS, we start with RTT of 10msec (which is still quite conservative).

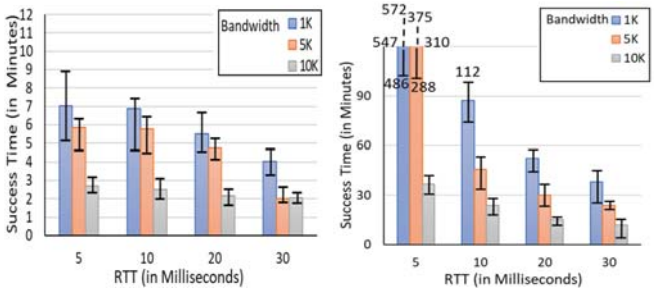
3) Mac OS: Figure 6-a shows the results of the attack against Mac OS. Similar to the Linux attacks, we assume a TTL of $30secs$ for the resource record in the OS-wide DNS cache. As we can see, overall time to success improves with increased bandwidth or RTT. Although the time to success is not prohibitive, the attack is slow. Thus, as we discussed earlier, we consider a case where the attacker also performs a DoS attack against the resolver with 15K packets per second (could be from the same attack machine or an external one). For ethical reasons, we set up our own resolver for this experiment so that we do not carry out a DoS



(a) Mac OS (without DoS) (b) Mac OS (with DoS)

Fig. 6: Median time to success on Mac OS without NAT

attack on part of the DNS infrastructure. The time to success improves dramatically, to a few minutes or lower under most configurations, as shown in Fig. 6-b. We note that more than 60% of the attacks succeeded in less than a minute for the configuration shown in Fig. 5.). With the DoS attack, the average number of trials to succeed drops dramatically to less than 5, since most of the real DNS requests are dropped by the resolver, providing a larger window to spoof responses.



(a) Windows (b) Ubuntu Linux

Fig. 7: Median time to success when client is behind NAT

B. Client behind NAT

Fig. 7-a and Fig. 7-b show the attack results against Windows and Ubuntu Linux when the client is behind a NAT (Mac OS behaved similar to Linux), respectively. As shown in the figures, in addition to RTT and bandwidth, the success time is also affected when we add NAT to the network topology.

Network congestion due to overflowing buffers within the network software stack is the main cause of packet loss which makes the attacks more challenging. Specifically, when the socket receive queue of the NAT receives packets at a rate that exceeds the router's renaming and forwarding capacity, the NAT starts dropping packets. In our case, since the attacker sends malicious response packets aggressively, we found that a fraction of these packets is not delivered to the client machine. As a result, we observe that the number of rounds until an attack is successful increases when NAT is present resulting in the higher success times. Moreover, on the client side, before a response packet gets processed by the DNS API `getaddrinfo()`, it is transmitted through different network queues until it reaches the OS-wide UDP socket receive queue. Since the IP stack filling the queue and the network driver that is draining the queue run asynchronously, packets might be dropped before they are even processed by DNS. This problem is especially apparent in the Mac attack results as shown in Fig. 6-a since the queue size is small (only 9216 Bytes) compared to Ubuntu Linux (212992 bytes).

VII. Attack Mitigation

Since existing defenses do not prevent the proposed attack, we explore a defense which requires only changes to the client DNS cache software, making it practical to deploy immediately through an OS patch. The defense first detects an attack using unique signatures (a large number of DNS replies with wrong TXID and local UDP port reservation), and then takes corresponding measures for mitigation.

Attack Detection. The detection module sniffs on UDP port 53 for all DNS traffic. After a DNS query is sent, a potential attack can be detected by observing a number of malicious responses with incorrect TXID. Once an attack is detected, a number of mitigations are possible, which we describe in the remainder of this section.

Attack Mitigation. The proposed mitigation strategies are two fold: (1) targeting the local malicious process; and (2) preventing the cache poisoning. To interfere with the malware process, the client OS notifies the user about the attack and the perpetrating process that reserves a large number of ports. Then depending on the user's preference or pre-configured security policy, the OS takes corresponding mitigation actions. A preferred action is to stop the malicious process, release the reserved ports, and then resend the same DNS query.

Verify Response to Interfere with cache poisoning. An additional mechanism we propose is to verify the responses received when an attack signature is detected. We consider two potential verification strategies: (1) Repeat request: since the likelihood of attack success in any round is small, we can accept a response only if it stays the same in successive lookups; (2) Verify by reverse lookup: we verify the IP address in the response DNS packet by sending a Pointer (PTR) DNS query. This query type is used to resolve an IP address to an FQDN. If the FQDN and the query name of the pending query do not match, then we can be more confident that an on-going attack is present (we notice that the PTR reply itself can also be spoofed though). We probed Alexa top 500 global websites by sending PTR DNS requests, and found that PTR queries have moderate support on today's Internet. We found that nearly 52% of the sites have PTR records and only 24% return an FQDN that contains the domain name in the query. Thus, this defense will work only opportunistically.

With these mechanisms, we suggest a less intrusive but more risky action of accepting the returned DNS response with the correct TXID but not cache it until/unless it is verified. By not caching the DNS response, at least we limit the damage by making sure that the DNS cache is not poisoned, allowing the cache to be used only when no foul play is suspected.

We tested the above defense on a number of operating systems including Windows, Mac OS, Ubuntu Linux as well as

on a NAT device (Linksys WRT3200ACM router running DD-WRT firmware). The implementation uses a co-located proxy (to enable portability across different operating systems), but ideally the defense would be realized directly by the DNS caching module itself in the OS. The defense mitigates all of our attacks: we were not able to poison the cache after running each attack for several hours with the defense deployed.

Other Recommendations. Additional mechanisms to hinder the attack include having the OS restrict the total number of open sockets to avoid the port reservation attack. Even though `ulimit` is supposed to limit the file descriptors of each user to 1024 or 4096, it does not seem to be enforced correctly on Mac or Linux at the moment. A second recommendation is to have the OS provide isolation among users of the same machine with each user having its own *dedicated DNS cache*. Isolation prevents a malicious user from poisoning the cache for other users as in the attack scenario with a public machine.

VIII. Conclusion

To conclude, we are the first to practically report, evaluate, and measure the client-side OS-wide DNS cache poisoning attack against Windows, Mac OS, and Linux operating systems. By understanding the specific OS implementations, we tailor the attacks against them individually and show that the attack can generally succeed in tens of seconds under realistic conditions. We hope that the lessons learned can help improve the future design and implementation of DNS and even other OS-wide caching systems.

Acknowledgments: Fatemah Alharbi is supported by Taibah University (TU) and the Saudi Arabian Cultural Mission (SACM). The work is partially supported by the National Science Foundation under grants No. CNS-1619391, CNS-1652954, and CNS-1618898.

References

[1] D. Atkins and R. Austein, "RFC 3833 - Threat Analysis of the Domain Name System (DNS)," https://tools.ietf.org/html/rfc3833, 2004.

[2] D. Kaminsky, "Black ops 2008: It's the end of the cache as we know it," *Black Hat USA*, 2008.

[3] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from dns rebinding attacks," *ACM Transactions on the Web (TWEB)*, vol. 3, no. 1, p. 2, 2009.

[4] A. Klein, "Bind 9 dns cache poisoning," *Report, Trusteer, Ltd*, vol. 3, 2007.

[5] "Multiple DNS implementations vulnerable to cache poisoning," http://www.kb.cert.org/vuls/id/800113, 2012.

[6] S. Weiler and D. Blacka, "Rfc 6840 - clarifications and implementation notes for dns security (dnssec)," https://tools.ietf.org/html/rfc6840, 2013.

[7] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee, "Increased dns forgery resistance through 0x20-bit encoding: security via leet queries," in *ACM CCS*, 2008, pp. 211–222.

[8] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "RFC 4035 - Threat Analysis of the Domain Name System (DNS)," 2005.

[9] S. M. Bellovin, "Security problems in the tcp/ip protocol suite," *ACM SIGCOMM CCR*, vol. 19, no. 2, pp. 32–48, 1989.

[10] P. Vixie, "Dns and bind security issues." in *Usenix Security*, 1995.

[11] A. Klein, "OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability," http://www.openbsdsupport.com.ar/books/OpenBSD_DNS_Cache_Poisoning_and_Multiple_OS_Predictable_IP_ID_Vulnerability.pdf., 2007.

[12] ——, "Windows DNS Server Cache Poisoning ," https://dl.packetstormsecurity.net/papers/attack/Windows_DNS_Cache_Poisoning.pdf, 2007.

[13] D. J. Bernstein, ""dns forgery"," https://cr.yp.to/djbdns/forgery.html, 2002, internet publication.

[14] ——, ""the dns random library interface"," https://cr.yp.to/djbdns/dns.html, 2008, internet publication.

[15] A. Hubert and R. van Mook, "RFC 5452 - Measures for Making DNS More Resilient against Forged Answers," 2009.

[16] M. Larson and F. Gont, "RFC 6056 - Recommendations for Transport-Protocol Port Randomization," https://tools.ietf.org/html/rfc6056, 2011.

[17] M. Larson and P. Barber, "RFC 4697 - Observed DNS Resolution Misbehavior," https://tools.ietf.org/html/rfc4697, 2006.

[18] A. Herzberg and H. Shulman, "Fragmentation considered poisonous, or: One-domain-to-rule-them-all. org," in *Communications and Network Security (CNS), 2013 IEEE Conference on*. IEEE, 2013, pp. 224–232.

[19] Y. Gilad, A. Herzberg, and H. Shulman, "Off-path hacking: The illusion of challenge-response authentication," *IEEE Security & Privacy*, vol. 12, no. 5, pp. 68–77, 2014.

[20] A. Herzberg and H. Shulman, "Security of patched dns," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 271–288.

[21] ——, "Vulnerable delegation of dns resolution," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 219–236.

[22] H. Shulman and M. Waidner, "Fragmentation considered leaking: port inference for dns poisoning," in *International Conference on Applied Cryptography and Network Security*. Springer, 2014, pp. 531–548.

[23] A. Klein, H. Shulman, and M. Waidner, "Internet-wide study of dns cache injections," in *IEEE INFOCOM*, pp. 1–9.

[24] R. Perdisci, M. Antonakakis, X. Luo, and W. Lee, "Wsec dns: Protecting recursive dns resolvers from poisoning attacks," in *IEEE/IFIP DSN*, 2009, pp. 3–12.

[25] "TLD Zone File Statistics," https://www.statdns.com/, online; accessed 24 July 2018.

[26] P. Hoffman and P. McManus, "DNS Queries over HTTPS," https://tools.ietf.org/html/draft-hoffman-dns-over-https-01, 2017.

[27] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman, "Specification for dns over transport layer security (tls)," Tech. Rep., 2016.

[28] F. Denis, "DNSCrypt," https://www.dnscrypt.org/, 2015.

[29] K. Schomp, M. Allman, and M. Rabinovich, "Dns resolvers considered harmful," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 16.

[30] T. Killalea, "RFC 3013 - Recommended Internet Service Provider Security Services and Procedures," 2000.

[31] R. Beverly, A. Berger, Y. Hyun *et al.*, "Understanding the efficacy of deployed internet source address validation filtering," in *ACM IMC*, 2009, pp. 356–369.

[32] R. Beverly, R. Koga, and K. Claffy, "Initial longitudinal analysis of ip source spoofing capability on the internet," 2013.

[33] P. Mockapetris, "State of IP Spoofing," https://spoofer.caida.org/summary.php, online; accessed 7 May 2018.

[34] T. Ehrenkranz and J. Li, "On the state of ip spoofing defense," *ACM Transactions on Internet Technology (TOIT)*, vol. 9, no. 2, p. 6, 2009.

[35] Z. Qian and Z. M. Mao, "Off-path tcp sequence number inference attack-how firewall middleboxes reduce security," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 347–361.

[36] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative tcp sequence number inference attack: how to crack sequence number under a second," in *ACM CCS*, 2012, pp. 593–604.

[37] "setrlimit(2) - Linux man page," https://linux.die.net/man/2/setrlimit, online; accessed 8 May 2018.

[38] "Mac OS X/Darwin man pages: setrlimit (2)," http://www.manpages.info/macosx/setrlimit.2.html, online; accessed 8 May 2018.

[39] "Top Sites in United States," https://www.alexa.com/topsites/countries/US, 2017.

[40] D. Kirkland, "Ubuntu Manpage: systemd-resolved.service, systemd-resolved - Network Name Resolution," http://manpages.ubuntu.com/manpages/bionic/man8/systemd-resolved.service.8.html.

[41] "dnsmasq," https://wiki.archlinux.org/index.php/dnsmasq, 2017.

[42] "Use of DNSSEC-ECDSA Validation for World (XA)," https://stats.labs.apnic.net/ecdsa/XA, online; accessed 25 July 2018.

[43] M. Brown, "The linux traffic control HOWTO," 2006, accessed May 2018 from http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html.

[44] D. Altman, D. Machin, T. Bryant, and M. Gardner, *Statistics with confidence: confidence intervals and statistical guidelines*. John Wiley & Sons, 2013.