DNS Poisoning of Operating System Caches: Attacks and Mitigations

Fatemah Alharbi, *Member, IEEE*, Yuchen Zhou, Feng Qian, *Member, IEEE*, Zhiyun Qian, *Member, IEEE*, and Nael Abu-Ghazaleh, *Senior Member, IEEE*

Abstract—The Domain Name System (DNS) is a protocol supporting name resolution from Fully Qualified Domain Names (FQDNs) to the IP address of the machines corresponding to them. This resolution process is critical to the operation of the Internet, but is susceptible to a range of attacks. One of the most dangerous attack vectors is DNS poisoning where an attacker injects malicious entries into the DNS resolution forcing clients to be redirected from legitimate to malicious servers. Typically, poisoning attacks target a DNS resolver allowing attackers to poison a DNS entry for all machines that use the compromised resolver. However, recent defenses protect resolvers substantially limiting these attacks. In this paper, we present a new class of DNS poisoning attacks targeting the client-side DNS cache, which is used in mainstream operating systems, circumventing defenses protecting resolvers. We implemented the attack on Windows, Mac OS, and Ubuntu Linux machines. 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. We also propose client-side mitigations and demonstrate that they can effectively mitigate the vulnerability.

Index Terms—DNS, NAT, cache poisoning, network security, microsoft windows, mac, Ubuntu linux

1 Introduction

The Domain Name System (DNS) protocol provides an integral service underlying the Internet: it provides a resolution service primarily of Fully Qualified Domain Names (FQDNs) (i.e., human-readable domain names such as `foobar.com`) to their corresponding Internet Protocol (IP) addresses [1]. 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. This hierarchy consists of 13 trusted root servers which are geographically widespread across the world, top-level domains (TLDs) (e.g., `.com` and `.edu`), and authoritative name servers (e.g., `foobar.com` and `ucr.edu`). DNS queries from clients are serviced using resolvers that can walk the DNS hierarchy to reach an authoritative name server that provides the answer to the DNS query. This resolution process can be slow because as it consists of several network round-trips. To improve performance, resolvers and end hosts use caching, exploiting locality across different hosts that share the cache.

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 vectors against DNS is the cache poisoning, where an attacker attempts to inject malicious DNS mappings to the cache of a DNS resolver [2], [3]. DNS is vulnerable to poisoning attacks because the process of resolving a domain name can take time: requests are sent to name servers which can be far away, leaving an open window of time for an attacker to inject a false response. More precisely, an attacker can poison the cache by impersonating authoritative name servers of a target domain [3], [4], [5]. The attack proceeds as follows: an attacker who wants to target the domain name *www.bank.com* waits until a victim resolver sends a DNS query for it. The attacker impersonates the legitimate DNS server by spoofing a response with a malicious IP. The malicious mapping provided by the attacker is cached by the resolver and all future queries for *www.bank.com* coming to the resolver will return the malicious IP address.

To protect against cache poisoning attacks on DNS resolver caches, Source UDP Port Randomization (SPR) [3], [6] 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

- *Fatemah Alharbi is with Computer Science Department, Taibah University, Yanbu 446412, Saudi Arabia. E-mail: fmhharbi@taibahu.edu.sa.*
- *Yuchen Zhou is with the College of Computer Science, Northeastern University, Boston, MA 02115 USA. E-mail: zhou.yuc@husky.neu.edu.*
- *Feng Qian and Nael Abu-Ghazaleh are with Computer Science and Engineering Department, University of California, Riverside, CA 94551 USA. E-mail: fengqian@umn.edu, nael@cs.ucr.edu.*
- *Zhiyun Qian is with the Computer Science and Engineering Department, University of Minnesota Twin Cities, Minneapolis, MN 55455 USA. E-mail: zhiyunq@cs.ucr.edu.*

Manuscript received 3 Dec. 2019; revised 15 July 2020; accepted 10 Sept. 2020. Date of publication 13 Jan. 2022; date of current version 9 July 2022.
The work of Fatemah Alharbi was supported by Taibah University (TU) and the Saudi Arabian Ministry of Education. This work was supported by National Science Foundation under Grants CNS-1619391, CNS1652954, and CNS-1618898. (Corresponding author: Fatemah Alharbi.)
Digital Object Identifier no. 10.1109/TDSC.2022.3142331

of effective cache poisoning attacks. Even though fundamentally secure DNS protocols such as *DNSSEC* [7] 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 a 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 [3], [6] and 0x20 [8] do not protect against this new attack. Even new proposals such as DNSSEC which rely on cryptography to completely close cache poisoning [1] 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 such as (1) Re-sending the same query and evaluating the response; (2) Using reverse lookup to verify that the response matches the expected domain of the query; and (3) Refusing to cache responses from this DNS server and switch to an alternative server. The defense requires only a client-side patch. We show that the defense mitigates the attack while maintaining the ability of the client to continue to resolve DNS requests.

Disclosure. We reported the attack to Apple, Microsoft, and Ubuntu. In response, Apple released a security update[1] fixing the `mDNSResponder` daemon, also known as Bonjour, that is used by the DNS Service Discovery API in Mac OS X (version 10.2 and later) and iOS operating systems. Ubuntu confirmed the vulnerability[2]. We understand that Microsoft is considering mitigations.

The paper makes the following contributions.

- We describe a new and dangerous DNS cache poisoning attack against client caches, which overcomes the challenges of source UDP port randomization. The techniques can defeat the challenge-response defenses that are mostly based on randomizing TXIDs and port numbers.
- We explore the attack when the victim is behind a NAT. We discover that many NAT boxes are fixed-port and/or port-preserving, enabling the attack to work without modification.
- We demonstrate the attack on three different operating systems: Microsoft Windows, Mac OS, and Linux Ubuntu machines. Each implementation introduces challenges, but we show that they are all addressable, enabling the attack to work with reliably. We measure the attack's effectiveness under various settings and show that the attacks are potent under realistic conditions. For instance, the attack typically succeeds in a few seconds on Windows.
- We propose a defense that relies on detecting the signature of the poisoning attacks which necessarily requires a large number of guesses. Once an attack is detected, we explore a number of reactions from the client that mitigate the attack. We experimentally show that the defense is highly efficient and effective mitigating the attack.

The remainder of the paper is organized as follows. Section 2 summarizes the related work. Section 3 provides the background and describes the threat model. The attack is introduced in Section 4. Tailored attack strategies and analysis are presented in Section 5. The attack is evaluated in Section 6. Section 7 proposes defense mechanism before concluding the paper in Section 8.

Readers are encouraged to watch a video demo illustrating our proposed attack on Windows:
https://www.youtube.com/watch?v=ulaccbjA6ZU

2 Background–Cache Poisoning Attacks

DNS cache poisoning is a dangerous class of attacks where an attacker injects a malicious mapping into a DNS cache to redirect communication to an adversarial server. This attack

1. https://support.apple.com/en-us/HT209446
2. https://bugs.launchpad.net/ubuntu/+source/systemd/+bug/1782225

remains a dangerous vulnerability: an Internet-scale measurement study on the vulnerability of DNS resolvers discovered that 92% of resolvers are vulnerable to at least one type of cache poisoning attack, despite advances in both attacks and defenses [9]. In this section, we overview the development and advances in cache poisoning attacks and the proposed mitigations for them, showing that our attack is not stopped by any of the existing defenses.

In 2007, Klein introduced a sophisticated cache poisoning attack against BIND 9 DNS resolvers [10] and Windows DNS servers [11]. At that time, the attack's entropy was completely derived from the TXID field in the DNS packet (the Transaction ID), and the implementation of the randomization algorithm for this field was predictable, facilitating the attack. Consequently, in 2008, Kaminsky presented another significant attack [3] against DNS resolvers which also bypasses the TXID based entropy used for authentication. The attack assumes both the UDP source and destination ports are fixed as 53. Indeed, Vixie already reported this potential vulnerability in 1995 [12], 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], [16], [17].

After this initial exploration of this attack, starting in 2011, several new cache poisoning attacks against resolvers were presented which have varying degrees of assumptions of the attack requirements and the network . For example, some attacks [18], [19] assume an attacker collaborates with a malicious script (e.g., Javascript in a browser) to poison the DNS cache of a resolver. 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. The same authors [20] propose a poisoning attack that exploits delegation of DNS resolution where intermediary network devices perform recursive lookups on behalf of the resolvers. In addition, they also show attack principles (but do not demonstrate the attack) [4], [21] that can poison the cache of a DNS resolver located behind a NAT. In contrast to these works, our attack is the first to target client caches, and therefore is not prevented by defenses such as DNSSEC as we discuss below.

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 [6], 0x20 encoding [8], random selection of authoritative name servers [15], [17], and adding random prefixes to domain names [22]. All these defenses attempt to make the space of packets from which a correct response must be selected larger, substantially increasing the attacker's difficulty in generating a valid response. Challenge-response defenses are vulnerable to MitM adversaries who can intercept the communication.

To protect against this type of eavesdropping attack, cryptographic defenses were proposed which include primarily DNSSEC [7] 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, the Internet Society organization reported that roughly 10% of a sample of size more than 735 million resolvers use DNSSEC validation [23]. Moreover, a recent study [24] discovered that 0.67%, 0.91%, and 0.91% of .com, .net, and .org Top Level Domains (TLDs) are signed. All these studies conclude that complete adoption of DNSSEC would prevent known DNS cache poisoning vulnerabilities, but a combination of partial deployment and permissive policies (e.g., accepting responses even if the DNSSEC signature does not match) lead to the current resolver vulnerabilities. Moreover, unless DNSSEC is extended to cover end clients, which introduces a substantial key distribution overheads, verification overheads, and is likely to infringe on usability, it does not prevent our attack. Recent defenses such as DNS over HTTPS [25], DNS over TLS [26], and DNScrypt [27], 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. It is also unclear if they will be used only by browsers due to the reliance on HTTPS.

A third defense alternative considers rethinking DNS implementation radically, resulting in different security properties. For example, Schomp *et al*. [28] 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. Currently, the security of patched operating systems relies mostly on the randomness of source ports. In our attack model, we expose vulnerabilities in the source port allocation algorithms used by three operating systems: Microsoft Windows, Mac OS, and Linux Ubuntu.

3 Attack Fundamentals and Threat Model

In this section, we set the table for the client-side DNS cache poisoning attack. We first describe the behavior of the OS-wide DNS cache of operating systems and then discuss the threat model.

3.1 OS-Wide DNS Caches

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.

When a client machine issues a DNS request, the request reaches the authoritative name server having the answer after traversing a chain of DNS servers/resolvers and different layers of caches. Some operating systems, e.g., Linux Ubuntu 16.04.6 LTS and earlier, are configured to use an external DNS server as their first resolver and uses its cache for future queries. Another alternative is to implement a client-side DNS cache internally. In this case, the OS first accesses the OS-wide cache and checks if the DNS record corresponding to the client's request is cached; if it is not cached, the OS forwards the request to the first external resolver in the Internet cloud; e.g., carrier's resolver. The latter implementation is vulnerable to client-side cache poisoning attacks. Our attack differs from other DNS cache poisoning attacks in that it targets the client-side cache, and therefore bypasses all known defenses that protect the resolvers.

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. We also verified that all applications first consult this cache before issuing an external DNS request on all OS versions we studied. Specifically, the OS-wide DNS cache is implemented by introducing a DNS system service, running in a separate process isolated from applications, that manages all the mappings, 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. Recently, Ubuntu started using the OS-wide DNS cache by default, but earlier distributions including Ubuntu 16.04 LTS have implementations that have to be optionally installed and enabled by users.

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, by using standard OS interfaces to display cached entries in the OS-wide DNS cache (e.g., `ipconfig /displaydns`, `sudo killall -INFO mDNSResponder`, and `sudo service nscd status` commands in Windows, Mac, and Ubuntu, respectively), 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 [3] relying on malicious records from the *additional section* does not apply to OS-wide DNS caches.

3.2 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 Section 4). This malware could be a malicious JavaScript running in the browser after an attacker 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 [29] is implemented [30]. 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 [31], [32], [33]. Another scenario where the attack is possible is one where two machines are located in the same network (e.g., enterprise or university network). It is common that a machine can spoof the IP address of the other since a packet does not pass through the ingress filter if it stays in the same network (we confirmed, after obtaining permission, this behavior in one enterprise and several university campuses).This threat model matches the threat model used in recent papers (e.g., off-path TCP injection attacks [34], [35]).

We also note another scenario where our attack applies: machines that have multiple users. One user can log into the machine, execute the attack causing the OS-wide DNS cache to be poisoned. When this user logs off, any subsequent user will be using the poisoned entries. We confirmed this attack scenario on several networks (with permission).

Another interesting threat model that we did not consider exploits the browser level caches: since some browsers (e.g., Firefox and Chrome) have their own DNS caches which are shared among different tabs/windows, the attack can also be applied using a malicious JavaScript piece of code assuming the success time is relatively small due to page visit time. In this scenario, the poisoning affects both the OS-wide and the browser-level DNS caches.

4 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.

4.1 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. In practice, before the unprivileged malware triggers the DNS request, it alerts the off-path attacker so that when the DNS request is sent, the off-path

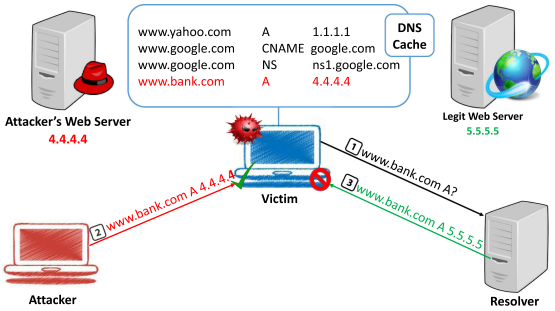


Fig. 1. High level attack overview.

attacker has started to respond by flooding the client with spoofed responses, maximizing the use of the vulnerable window between the issue of a DNS request and the receipt of a valid response. The OS-wide DNS API accepts the first correct response: a response with both: (1) a matching port number so that it is received correctly; and (2) a matching TXID field. If a correct response from the attacker is received before the legitimate response from the DNS resolver, the client simply accepts this response and caches it in its OS-wide DNS cache (i.e., the legitimate response is discarded). 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.

4.2 Attack Scenarios

The attack requires a malicious user level program to execute on the victim machine to initiate the DNS request. 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. However, without administrator privileges, the attacker may wait for TTL to expire (which is the case with macOS and Linux Ubuntu), or evict the cache by requesting a large number of DNS resolutions to get older entries removed from the cache, either of which slows down the attack. To confirm the practicality of the attack in such a scenario, we conducted an empirical study on shared public machines (such as the ones found in student labs and libraries) in four different universities (we do not disclose their names for ethical reasons). For all of the tested networks, we found that the shared machines share the same private address space (e.g., `192.168.x.x`) and connect to a single default gateway. We also found that they all use the corresponding university's DNS resolver to perform the DNS resolution process on their behalf. After obtaining permission from the system administrators to conduct the experiments then clearing the cache, we confirmed that the universities networks 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.

4.3 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. DNS primarily uses the UDP protocol to transmit packets[3]. In our attack, we assume that all DNS traffic between victim, the source, and resolver, the destination, is over UDP, which is a valid assumption because DNS `A` packet size is usually smaller than 512 bytes. The 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 only challenge in guessing the correct packet fields 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, including Simple Port Randomization, Simple Hash-Based Port Selection (which is implemented by Linux), Double-Hash Port Selection, and Random-Increment Port Selection algorithms [16]. Many are proposed specifically to defeat port prediction attacks, which means the port allocated each time for a new socket will appear to be random. Prior work [19], [21] shows that these algorithms are not efficient and can be reverse engineered and guessed. Lacking documentation, it was unclear which algorithm is used in Windows or Mac OS. 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. We did find that older editions of Windows such as Windows 2000 and Windows XP assign ports to connections sequentially, but this no longer is the case in modern versions of these Operating Systems.

A basic building block of our attack is the ability to predict or infer the source UDP port of a DNS request. Based

3. TCP is used in cases where the data size of requests and responses exceeds 512 bytes; see RFC 1035 [36], but there is an extension to the DNS protocol that allows increasing payload size in UDP datagram (RFC 6891 [37]). TCP also used in transferring Zone data because it is reliable.

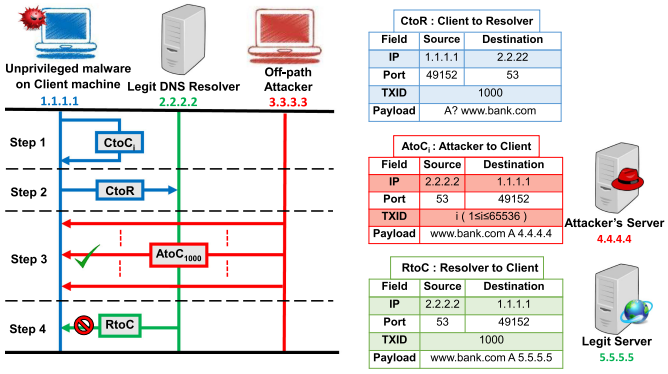


Fig. 2. Design of client-side DNS cache poisoning attack.

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 [38]. Likewise, Ubuntu Linux has a default limit of 4096 file descriptors for each process which also can be raised to meet the attack requirements [39]. 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 Fig. 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 $AtoC_1$, $AtoC_2, \ldots, AtoC_{65536}$, 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 rounds (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.

5 Tailored Attack Strategies and Analysis

In this section, we consider OS-specific attack strategies, and also consider whether the client is behind a NAT router. We organize the discussion into first attacks without considering NAT, and then the situation where the client is behind a NAT router.

5.1 Basic Attack Scenarios (Without 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. In other words, the communications between the client, resolver, and attacker are direct and not translated. This setting is fairly common as many networks do not use NAT. Moreover, the attacker can be a legitimate user on the same organizational network with the victim (i.e., on the same private network) or on a public WiFi wireless network in the same organization. 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, creating a legitimate response appears to be a challenging task as there is really only an attack window equivalent of a Round-Trip Time (RTT) between the client and resolver. Even with a high bandwidth, the attack has a fairly low probability of success. Assuming an RTT of 5msec between a client and a nearby resolver, and further assuming 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 time out before being able to launch the next round of attack on the same domain. However, we discover that *OS implementation specifics* and *real-world TTL/network latency* make our proposed attack highly feasible. We next detail attacks against three OSes:

5.1.1 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). Although this behavior means that the attacker only has five chances to guess the TXIDs before each invocation of `getaddrinfo()`, we found that this behavior does not limit the attack since the attacker can

TABLE 1
The Caverage 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

always call `getaddrinfo()` multiple times until the attack succeeds. More precisely, the strategy works across multiple invocations as the attacker keeps invoking `getaddrinfo()`, making the attack easier since the true response for each new invocation does not arrive in time before the request resets.

5.1.2 Mac OS

For Mac OS, we find that the DNS system service continues to accept DNS responses until receiving a response with a matching TXID. If no response is found before the timeout, the service automatically retransmits the request and continues to wait for the correct response to arrive. This implementation results in the attack window being only a single RTT before the legitimate response is received and the response cached. As a result, the attack window is somewhat limited: if we can send 10K spoofed messages per second, and $RTT = 5msec$, then the number of chances is 50. Trying the attack again with another `getaddrinfo()` results in the value being returned from the cache. Accordingly, 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. We conducted measurements to show the TTLs of the top 10 global websites based on Alexa [40]. 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 [40]. 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 1, the RTTs are indeed relatively high.

5.1.3 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 [41]. 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 Section 4.3 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 [42], 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.

5.2 Client Behind NAT Attacks

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 [43]. More specifically, the benefits of NAT include: (1) allowing administrators to assign private IP addresses to machines without having globally assigned addresses; and (2) hiding the internal structure of a private network from the outside. Specifically, the NAT router's external facing IP is a valid static IP address. Since the machines behind the NAT do not have a globally resolvable address, the NAT translates their internal address on packets they send to the Internet to its external address [43].

To be able to resolve incoming packets (which all use the IP address of the NAT router) to the correct internal machine, NAT uses ports to keep track of connections belonging to different machines. In particular, the router also assigns an external source port which could be different than the source port selected by the operating system of the client who initiated the connection and keeps the mapping between the ports in a mapping table. When an incoming packet comes to a particular port, the NAT router replaces the destination IP and port number with the IP address and port number of the internal port. 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 popular 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, to use the default settings of the client's ISP, we found that 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 DNS settings to an alternative DNS server (e.g., 8.8.8.8 for Google public DNS), 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 [23].

3. NAT Manually Configures DNS. If the client uses DHCP but NAT uses a manually configured DNS server, NAT assigns a fixed source port to DNS sessions to that server.

4. Both Client and NAT Manually Configure DNS. Similar to Case 2, 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 modified to attack a client behind NAT. For Case 1, we can bootstrap our attack by using principles proposed by Herzberg *et al.* [4], [21] 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.

6 Evaluation

In this section, we conduct measurements on real systems to assess the effectiveness of the end to end attack in realistic settings for all our attack scenarios. The attack is successful within a reasonable time window against all these operating systems confirming that this is a dangerous vulnerability.

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. The malware runs in user space without privileges.

Network Configuration. We use two different network topologies mirroring those shown in Fig. ??a (without NAT) and Fig. ??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 Section 5). We also verified the attack on fixed-port NATs (case 3).

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` [44]. 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.

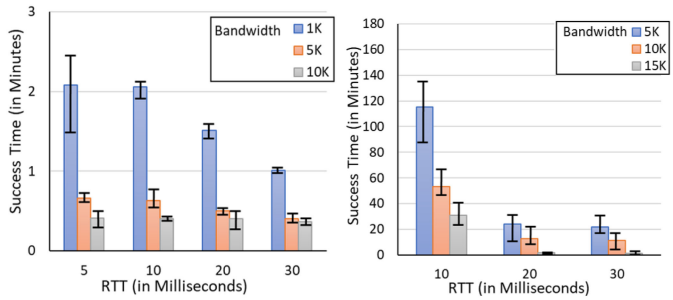


Fig. 3. Median time to an attack success without NAT.

Experimental Details. 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 [45] to calculate the *rank order* of the upper and lower bounds instead of their actual values.

6.1 Results of Attacks Without Considering NAT

Windows. Fig. 3a 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. 4, even though the number of rounds 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 msec after 5 spoofed responses are received. As discussed in Section 5.1, in theory, the two metrics should not be dependent on RTT and bandwidth so long as at least 5 spoofed responses always arrive before the authentic response. In practice, however, we observe that rarely ($< 0.1\%$ of the rounds) the authentic response arrives within the first 5 responses due to network jitter, especially as RTT gets larger. In general, 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. However, there are effects that interfere with the effective bandwidth of the attack. For example, we observe that the legitimate responses from previous queries sometimes consume some of the 5 slots available before a request is reset. For

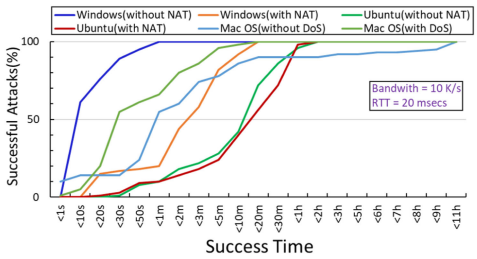


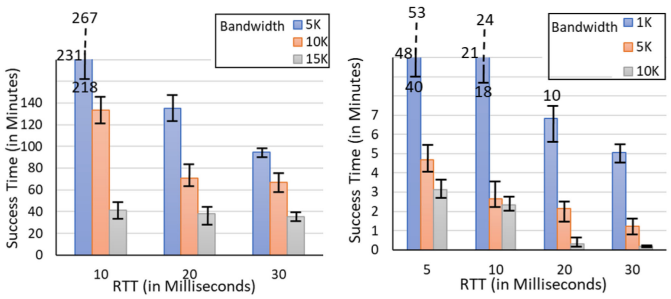
Fig. 4. Histogram of successful attacks over Time.

example, the `getaddrinfo()` may receive two authentic responses from the previous invocation (both with the same TXID), and it will only process the next three spoofed responses.

Ubuntu Linux. Fig. 3b 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 Section 5.1). Thus, for each failed round, the attacker waits for 30 seconds before the next round. As shown in the figure, the attack time-to-success also depends on RTT and bandwidth.

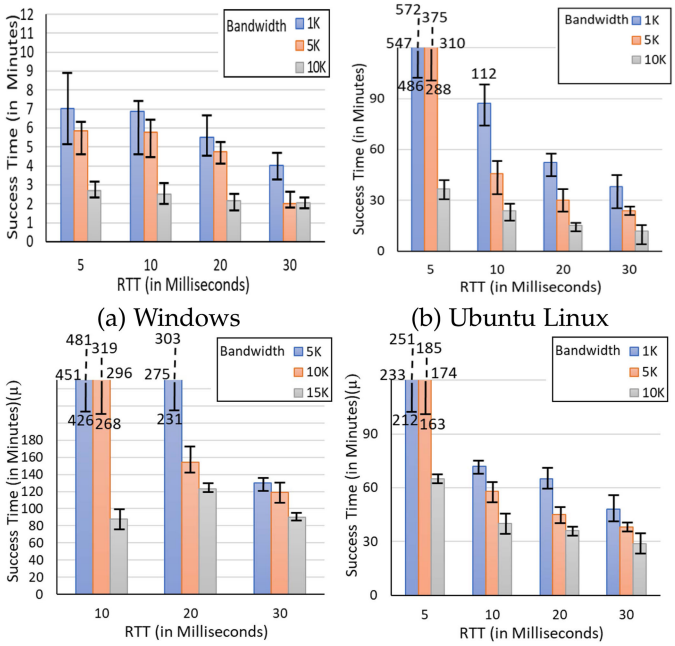
Note that in this scenario (as well as MacOS), every round waits until the TTL expires before it is initiated. Thus, the response to the query will miss the caches and likely go through the full resolution step, or hit a cache upstream, resulting in a large delay until the authentic response is received. We measured this delay to be on average 92msec (See Table 1). Thus, for the attack on Ubuntu and Mac OS, we start with RTT of 10msec (which is still quite conservative). We excluded the cases where $bandwidth = 1K/sec$ and $RTT = 5msecs$ because it is extremely challenging to a get a successful attack. For instance, we ran an experiment for 1 day using $bandwidth = 15K/sec$ and $RTT = 5msecs$ with no successes. Furthermore, in an experiment where $bandwidth = 1K/sec$ and $RTT = 30msecs$, we got a successful attack after 15Hrs.

Mac OS. Fig. 5a 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, we consider a case where the attacker also performs a DoS attack against the resolver with 15K packets per second. For ethical reasons, we set up our own resolver for this experiment to



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

Fig. 5. Median time to success on Mac OS without NAT



(a) Windows (b) Ubuntu Linux

(c) Mac OS (without DoS) (d) Mac OS (with DoS)

Fig. 6. Median time to success when client is behind NAT.

avoid affecting other users. The time to success improves dramatically, to a few minutes or lower under most configurations, as shown in Fig. 5b. We note that more than 60% of the attacks succeed in less than a minute for the configuration shown in Fig. 4.). The average number of rounds to succeed drops dramatically to less than 5, since most of the real DNS responses are dropped by the resolver, providing a larger window to spoof responses.

Packet loss due to congestion 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. Moreover, on the client side, before a response packet gets processed by the DNS API `getaddrinfo()`, it is transmitted through different layers of network queues until it reaches the OS-wide UDP socket receive queue. Since the IP stack that is filling the queue and the network driver that is draining the queue run asynchronously, there is a high probability of packet loss before reaching the DNS API. This problem is also apparent in Mac attack results as shown in Fig. 5a since the queue size is small (only 9216 Bytes) compared to Ubuntu Linux (212992 bytes).

Client Behind NAT. Fig. 6 shows the results when the client is behind a NAT: the time to success increases when we add NAT to the network topology.

6.2 Analytical Model and Validation

We model the probability of success as a geometric process with the constraints imposed by each Operating System (e.g., the 5 response limit in Windows). We compare these results to measurement results regarding the number of rounds before a successful attack. We note that the

analytical model is approximate because it does not capture network effects or other effects such as the legitimate responses competing with the attack responses particularly in Windows; it is difficult to model these effects because they vary with background traffic and implementation parameters such as buffer sizes.

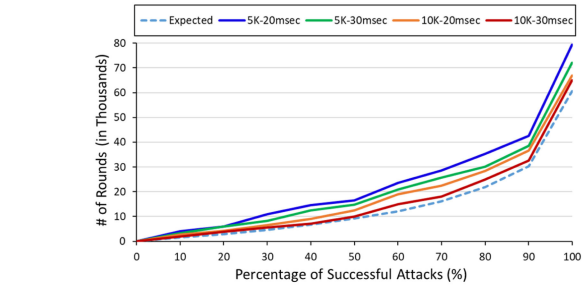
Analytical Model. For a single spoofed reply, the probability of making a correct guess for the TXID is 1 out of 2^{16} (2^{16} is the TXID field range). The probability of failure of the attack in response to a single invocation of DNS API (e.g., `getaddrinfo()`) is determined as follows: $P(failure) = \frac{2^{16}-z}{2^{16}}$, where z is the number of guesses up to a maximum of 2^{16}. We found that the value of z differs based on the DNS API `getaddrinfo()` of the different operating systems. 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 determined by the cumulative distribution function of the geometric distribution and is: $P(X \leq x) = 1 - P(failure)^x$.

The average number of rounds 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 takes as short as 2msec, which means that the average time to succeed will be as short as $2msec * 13K = 26$ seconds.

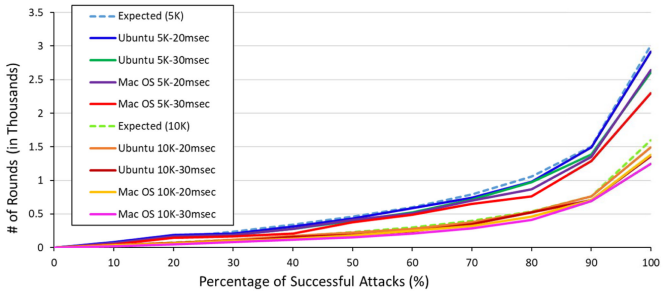
For Mac and Linux, z is determined by the bandwidth of the off-path attacker to the client and the RTT of the DNS resolution (which determines when the legitimate response is received). For example, pessimistically assuming a low RTT of 5msec, and an attacker bandwidth of 10K spoofed messages per second, 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 rounds 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 1 which is 92msec, z is 920 packets, leading to a success rate of approximately 1.4% and an average number of rounds 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.

Model Validation. To validate the analytical model, we conducted a measurement study of the attack on all three operating systems. For each Operating System, we show measurement results for different cases (i.e., when bandwidth from attacker to client is 5K and 10K and when RTT between DNS resolver and client is 20msec and 30msec); we use these parameters in the analytical model. For each experiment, 200 successful attacks are conducted to provide the measurement data. We show a histogram of the number of rounds to succeed for individual experiments to provide and compare that to the predicted percentage of successful attacks at each point.

Figs. 7a and 7b show that the results of the analytical model conform with the experimental data. For Windows, as shown in Fig. 7a, we find that we need tens of thousands of rounds to obtain a probability of success higher than 50%. The analytical model underestimates the number of required rounds due to network effects such as congestion when the attack traffic is present, which especially affect Windows due to the 5 packet response limit for each request. For Ubuntu and Mac OS, as shown in Fig. 7b, we see that the number of rounds in our measurements follow the analytical model closely. This is because the rate of retries is slower (because we have to wait for the DNS cache value to expire after the legitimate response is received), leading to less congestion.



(a) Windows



(b) Ubuntu and Mac OS

Fig. 7. Experimental and Analytical Results (# of Rounds). The figure shows that x% of the experiments (x is the value on the x-axis) succeed in the number of rounds shown on the y-axis. Since the success of the experiment is geometrically distributed, the "least lucky" 10% of the experiments do require almost double the number of rounds to succeed as the other 90%. This behavior is consistent with the analytical model which follows the same shape of the curve.

7 Attack Mitigation

We focus on client side defenses since they require only changes to the client software, and can therefore be deployed immediately through an OS patch. The defense first detects an attack using its unique signature (multiple DNS replies with wrong TXID), and then takes corresponding measures for mitigation.

7.1 Attack Detection

An overview of the proposed defense is presented in Fig. 8. The detection module is implemented as a proxy server for flexibility of evaluation against different Operating Systems; in a real deployment it can be integrated directly with the DNS service. The module sniffs on UDP port 53 for all DNS traffic. After a DNS query is sent in (1), an attack is detected in step (2) when observing responses with incorrect TXID. Once an attack is detected, we send a signal to the OS in step 3 on the figure, triggering mitigations. We can

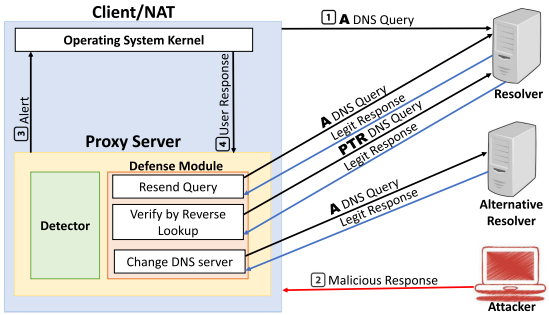


Fig. 8. Defense model.

optionally implement a local sanity check on the system to see if there is suspicious behavior; for example, we check if there are processes reserving a suspiciously large number of ports and clean those up. This step is optional because it may lead to collateral damage if a legitimate process uses a large number of ports. It is also possible to integrate this check as part of the system call to open a new socket to limit processes from opening an sockets when suspicious behavior is evident.

At the DNS service level (step 4), different reaction strategies are possible (shown under the defense module in the figure). A paranoid strategy that simply drops all responses can lead to Denial of Service as the attacker prevents legitimate responses; thus, it is important to find reaction strategies that can protect the system, but also allow to continue to function. We describe the strategies in the remainder of this section.

7.2 Attack Mitigation Strategies

The proposed mitigation strategies leverage the optional step of releasing the port reservations of the suspected malicious processes. They are three fold:

1. Repeat Query. The first mitigation strategy is to resend the DNS query again. Since the second request uses a different port number (assuming that we released the sockets as part of the defense), the success probability is independent of the success probability in the original request. If a different response is received (that is, we successfully poisoned one out of two), a third query is sent and the majority response cached.

In this case, the attack will succeed if we poison either two queries in a row, or two out of three. This would make the probability of a successful attack goes from P to the probability of either poisoning the two first queries (which is p^2), or poisoning one of the first two, and the third (which is $2 \cdot p(1-p) \cdot p$). Adding the two terms and simplifying, we get a probability of defeating the defense equal to $3p^2 - 2p^3$. For Windows, the probability of success would be 1.73×10^{-8}. By considering our analytical model (see Section Section 6.2), the number of rounds would increase to be more than 57 million, which would require many years for the attack to succeed.

2. Verify Response Using Reverse Lookup. The module verifies 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 a 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, but the probability of success on both the reply to both the A and PTR queries is very small). 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, or with support from websites and public servers to maintain accurate PTR resolutions.

3. Switch DNS Server. The last class of defense we explore is to choose an alternative DNS server, in combination with strategy 1, which is to resend the request. This requires the attacker to also guess the DNS server, further reducing the probability of success.

On top of these mechanisms, we suggest a less intrusive action (but potentially 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. In our scenario with the malware, the malware receives the spoofed response, but is unable to poison the entry for any other applications.

TABLE 2
Defense Experiments

Operating System	# of Attacks (without defense)	# of Attacks (with defense)
Windows	1705	0
Ubuntu Linux	152	0
Mac OS	18	0

7.3 Empirical Defense Evaluation

We tested the above defense on a number of operating systems including: Microsoft Windows 10, MacOS Sierra, and Ubuntu 17.04 without a NAT box. However, we also tested the defense on Linksys WRT3200ACM router running DD-WRT firmware and confirmed that the defense works efficiently. The attacker's machine runs Ubuntu Linux 16.04 LTS. We used RTT = 20msec and the attacker's throughput is 10K/s (i.e., 7.4Mbps when malicious packet size is 97Bytes). In our implementation, we considered two strategies: repeat query and verify by reverse lookup. We ran the attack continuously over a twenty-four hour period collecting results for the cases with and without the defense. In the first 12 hours, we used the first strategy while in the second half we used the second while continuously attempting to launch the attack (i.e., using the attack scripts that succeeded without the defense). As shown in Table 2, we did not observe any successes during this period. For reference, we recorded 1705, 152, and 18 successful attacks on Windows, Ubuntu Linux, and Mac OS, respectively, in a 12 hour period when the defense is not 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.

7.4 Disclosure

We reported the attack and our proposed defense to all three vendors. We report their responses next.

Apple MacOS. Apple responded and released a security update to fix the `mDNSResponder` daemon that is used by the DNS Service Discovery API in Mac OS X (version 10.2 and later) and iOS operating systems. We conducted experiments to confirm the mitigation. Specifically, we checked the DNS cache size and found that the number of entries has increased from 5076 to 7500. In addition, we were able to reserve only up to 30K file descriptors instead of 100K. We found that these two changes added more burden to get a successful attack in a reasonable time. In addition, we found that the `mDNSResponder` drops our spoofed packets if we send them aggressively (i.e., the time between trials is $\approx$ 10msec), interfering with our ability to fill the cache. If we slow down (assuming we sleep for 20msec between trials), filling the cache takes a considerable amount of time: in our measurements the process took $\approx$ 85 seconds. Although we managed to fill-in the cache in this time, we, however, encountered a significant problem. Most of the spoofed cached entries are stored as failed lookups; thus, when another successful DNS lookups happens, the new entries take the place of the old failed ones. As a result, filling-in the cache is impossible.

Linux Ubuntu. Ubuntu developers confirmed the vulnerability and considered disabling the OS-level DNS cache used by the default DNS service `systemd` and the DNS service of the `dnsmasq` library. We conducted measurements and confirmed that the cache is turned off by default and we could not get a successful attack.

Microsoft Windows. We understand that Microsoft is still considering mitigations. However, we did not find evidence about any kind of deployed defenses and are still able to launch the attack successfully with the latest patches.

8 Conclusion

This paper identifies and evaluates a new client-side OS-wide DNS cache poisoning attack against Windows, Mac OS, and Linux OSes. The attack targets the OS level DNS cache, a client-side cache supported by most modern OSes. We tailor the attacks to work against each OS individually taking into account the specifics of each implementation. The attack succeeds in as little as tens of seconds under realistic conditions. We also build an analytical model of the expected number of rounds for the attack to succeed for both Windows and Ubuntu and validate the model against the experimental observations. Finally, we propose a defense that requires only a patch of the OS of the client device and show that it can mitigate the attack.

Acknowledgments

We appreciate the insightful comments from Jie Chang, an employee at eSafe NET company, who helped us in reverse engineer the Windows DNS service library. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the funding agencies.

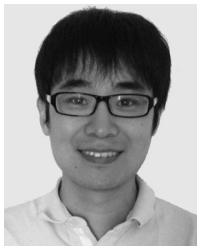
References

[1] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "RFC 4035 - Threat analysis of the domain name system (DNS)," 2005.
[2] D. Atkins and R. Austein, "RFC 3833 - threat analysis of the domain name system (DNS)," 2004. [Online]. Available: https://tools.ietf.org/html/rfc3833
[3] D. Kaminsky, "Black OPS 2008: It's the end of the cache as we know it," *Black Hat USA*, 2008.
[4] 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, Sep./Oct. 2014.
[5] A. Klein, "OpenBSD DNS Cache poisoning and multiple O/S predictable IP ID vulnerability," 2007. [Online]. Available: http://www.openbsdsupport.com.ar/books/OpenBSD_ DNS_Cache_Poisoning_ and_Multiple_OS_ Predictable_IP_ID_Vulnerability.pdf.
[6] "Multiple DNS implementations vulnerable to cache poisoning," 2012. [Online]. Available: http://www.kb.cert.org/vuls/id/800113
[7] S. Weiler and D. Blacka, "RFC 6840 - clarifications and implementation notes for DNS security (DNSSEC)," 2013. [Online]. Available: https://tools.ietf.org/html/rfc6840
[8] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee, "Increased DNS forgery resistance through 0x20-bit encoding: Security via leet queries," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 211–222.
[9] A. Klein, H. Shulman, and M. Waidner, "Internet-wide study of DNS cache injections," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[10] A. Klein, "Bind 9 DNS cache poisoning," *Report, Trusteer, Ltd*, vol. 3, 2007.
[11] A. Klein, "Windows DNS server cache poisoning," 2007. [Online]. Available: https://dl.packetstormsecurity.net/papers/attack/Windows_DNS_Cache_Poisoning.pdf
[12] P. Vixie, "DNS and BIND security issues," in *Proc. 5th Conf. USENIX UNIX Secur. Symp.*, 1995, Art. no. 19.
[13] D. J. Bernstein, "DNS forgery," 2002, [Online]. Available: https://cr.yp.to/djbdns/forgery.html
[14] D. J. Bernstein, "The DNS random library interface," 2008. [Online]. Available: https://cr.yp.to/djbdns/dns.html
[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," 2011. [Online]. Available: https://tools.ietf.org/html/rfc6056
[17] M. Larson and P. Barber, "RFC 4697 - observed DNS resolution misbehavior," 2006. [Online]. Available: https://tools.ietf.org/html/rfc4697
[18] A. Herzberg and H. Shulman, "Fragmentation considered poisonous, or: One-domain-to-rule-them-all. org," in *Proc. IEEE Conf. Commun. Netw. Secur.*, 2013, pp. 224–232.
[19] H. Shulman and M. Waidner, "Fragmentation considered leaking: Port inference for DNS poisoning," in *Proc. Int. Conf. Appl. Cryptography Netw. Secur.*, 2014, pp. 531–548.
[20] A. Herzberg and H. Shulman, "Vulnerable delegation of DNS resolution," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2013, pp. 219–236.
[21] A. Herzberg and H. Shulman, "Security of patched DNS," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2012, pp. 271–288.
[22] R. Perdisci, M. Antonakakis, X. Luo, and W. Lee, "WSEC DNS: Protecting recursive DNS resolvers from poisoning attacks," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2009, pp. 3–12.
[23] "Use of DNSSEC-ECDSA validation for world (XA)," Accessed: Jul. 25, 2018. [Online]. Available: https://stats.labs.apnic.net/ecdsa/XA
[24] "TLD Zone File Statistics," Accessed: Jul. 24, 2018. [Online]. Available: https://www.statdns.com/
[25] P. Hoffman and P. McManus, "DNS queries over HTTPS," 2017. [Online]. Available: https://tools.ietf.org/html/draft-hoffman-dns-over-https-01

[26] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman, "Specification for DNS over transport layer security (TLS)," *Tech. Rep.*, 2016.
[27] F. Denis, "DNSCrypt," 2015. [Online]. Available: https://www.dnscrypt.org/
[28] K. Schomp, M. Allman, and M. Rabinovich, "DNS resolvers considered harmful," in *Proc. 13th ACM Workshop Hot Top. Netw.*, 2014, Art. no. 16.
[29] T. Killalea, "RFC 3013 - recommended internet service provider security services and procedures," 2000.
[30] R. Beverly, A. Berger, Y. Hyun, and K. Claffy, "Understanding the efficacy of deployed internet source address validation filtering," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas.*, 2009, pp. 356–369.
[31] R. Beverly, R. Koga, and K. Claffy, "Initial longitudinal analysis of IP source spoofing capability on the internet," 2013.
[32] P. Mockapetris, "State of IP spoofing," Accessed: May 07, 2018. [Online]. Available: https://spoofer.caida.org/summary.php
[33] T. Ehrenkranz and J. Li, "On the state of IP spoofing defense," *ACM Trans. Internet Technol.*, vol. 9, no. 2, 2009, Art. no. 6.
[34] Z. Qian and Z. M. Mao, "Off-path TCP sequence number inference attack-how firewall middleboxes reduce security," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 347–361.
[35] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative TCP sequence number inference attack: How to crack sequence number under a second," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 593–604.
[36] P. Mockapetris, "RFC 1035 - domain names - implementation and specification," 1987. [Online]. Available: https://www.ietf.org/rfc/rfc1035.txt
[37] J. Damas, M. Graff, and P. Vixie, "RFC 6891 - extension mechanisms for DNS (EDNS0)," 2013. [Online]. Available: https://tools.ietf.org/html/rfc6891
[38] "setrlimit(2) - Linux man page," Accessed: May 08, 2018. [Online]. Available: https://linux.die.net/man/2/setrlimit
[39] "Mac OS X/Darwin man pages: Setrlimit (2)," Accessed: May 08, 2018. [Online]. Available: http://www.manpages.info/macosx/setrlimit.2.html
[40] "Top sites in United States," 2017. [Online]. Available: https://www.alexa.com/topsites/countries/US
[41] D. Kirkland, "Ubuntu Manpage: Systemd-resolved.service, systemd-resolved - network name resolution." [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man8/systemd-resolved.service.8.html
[42] "dnsmasq," 2017. [Online]. Available: https://wiki.archlinux.org/index.php/dnsmasq
[43] P. Srisuresh and M. Holdrege, "RFC 2663 - IP network address translator (NAT) terminology and considerations," 1999. [Online]. Available: https://tools.ietf.org/html/rfc2663
[44] M. Brown, "The linux traffic control HOWTO," 2006. accessed May, 2018 from http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html
[45] D. Altman, D. Machin, T. Bryant, and M. Gardner, *Statistics with Confidence: Confidence Intervals and Statistical Guidelines*. Hoboken, NJ, USA: Wiley, 2013.

Fatemah Alharbi (Member, IEEE) received the PhD degree from the University of California, Riverside, in 2020. She is currently an assistant professor with the Computer Science Department, Taibah University, Yanbu, Saudi Arabia. Her research interests include cybersecurity, network and system security, Internet of Things (IoT), applied cryptography, security and privacy-related real-world measurements and analysis.

Yuchen Zhou received the bachelor's degree in information security from the Chengdu University of Information Technology, in 2017 and the master's degree in cybersecurity from Northeastern University, in 2019. He is a security software engineer with Medtronic plc. His work focuses on the security of surgical robotics system, and he is also interested in IoT and system security.



Feng Qian (Member, IEEE) received the bachelor's degree from Shanghai Jiao Tong University and the PhD degree from the University of Michigan. He is currently an assistant professor with the Computer Science and Engineering Department, University of Minnesota. His research interests include cover the broad areas of mobile systems, VR/AR, computer networking, and system security.

Zhiyun Qian (Member, IEEE) received the PhD degree from the University of Michigan, Ann Arbor, in 2012. He is the Everett and Imogene Ross associate professor with the University of California, Riverside. His research interests are on system and network security, including vulnerability discovery, side channel analysis, applied program analysis, system building, and measurement of real-world security problems. He is a recipient of the NSF CAREER Award in 2017, Applied Networking Research Prize from IRTF in 2019, and Facebook Internet Defense Prize Finalist in 2016.

Nael Abu-Ghazaleh (Senior Member, IEEE) received the PhD degree from the University of Cincinnati, in 1997. He is currently a professor with the Computer Science and Engineering Department and the Electrical and Computer Engineering Department, the University of California, Riverside, CA, USA. His research interests include computer architecture support for security, parallel discrete event simulation, and networking and mobile computing.