Fragmentation Considered Leaking: Port Inference for DNS Poisoning

Haya Shulman and Michael Waidner

Fachbereich Informatik
Technische Universität Darmstadt
Darmstadt, Germany
{haya.shulman,michael.waidner}@cased.de

Abstract. Internet systems and networks have a long history of attacks by off-path adversaries. An off-path adversary cannot see the traffic exchanged by the legitimate end points, and in the course of an attack it attempts to impersonate some victim by injecting spoofed packets into the communication flow. Such attacks subvert the correctness and availability of Internet services and, among others, were applied for DNS cache poisoning, TCP injections, reflection DDoS attacks.

A significant research effort is aimed at hardening client systems against off-path attacks by designing challenge-response defences, whereby random challenges are sent with the request and the responses are validated to echo the corresponding values.

In this work we study the security of a standard and widely deployed challenge-response defence port randomisation, and show that off-path attackers can *efficiently* and *stealthily* learn the ports selected by end systems.

We show how to apply our techniques for DNS cache poisoning. We tested our attacks against standard and patched operating systems and popular DNS resolvers software. Our results motivate speeding up adoption of cryptographic defences for DNS.

Keywords: Challenge-response defences, DNS cache poisoning, fragmentation.

1 Introduction

Cryptography has known decades of research with numerous schemes, however, very few of those results are actually used in practice. Most Internet traffic is still not cryptographically protected, and basic systems and protocols, such as routing and naming, that constitute foundations of the Internet, are not protected. There is some (albeit limited) deployment of cryptography, mainly for protection of web traffic, e.g., based on the CAIDA dataset of 3 million packets [1] we found that only about 6% of the TCP traffic is cryptographically protected with SSL/TLS.

Currently, most systems deploy challenge-response defences, which provide security guarantees against off-path adversaries. Unlike a man-in-the-middle (MitM) adversary, an off-path adversary cannot observe nor modify legitimate packets exchanged between other parties. Off-path adversaries typically launch attacks by transmiting packets that contain a *spoofed* (fake) source IP address - impersonating some legitimate party; see attacker model in Figure 1. Spoofed packets are used in many attacks, most notably, in cache poisoning and Denial of Service (DoS) attacks. Significant efforts are invested to enforce ingress filtering, [RFC3704], in order to prevent spoofing. However, IP spoofing is still possible via many ISPs and networks, [2].

Challenge-response authentication provides means to distinguish between packets sent from spoofed source IP addresses and packets exchanged between legitimate communication end-points. In order to authenticate a response from a server, a client sends a random *challenge* within the request, which the server echoes in the *response*. Since an off-path attacker cannot eavesdrop on the packets exchanged between the server and the client, it appears that it would have to guess the challenge. Thus, a challenge, selected at random from a large distribution, should suffice to prevent an off-path attacker from crafting a response packet with valid challenge values.

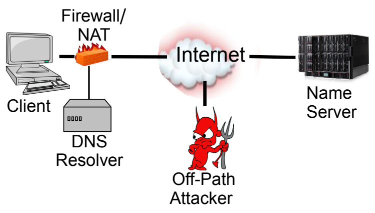

Fig. 1. Off-Path Attacker Model

The security of most Internet services, e.g., email, web surfing, DNS, peer-to-peer applications, relies on challenge-response mechanisms, mainly as part of the underlying *transport* and *application* layer protocols. A popular defence at the transport layer, supported by vast majority of the systems, is port randomisation, whereby the response is validated to have arrived on the same port from which the request was sent. Challenge-response defences in the application layer include widely-used web-security mechanisms based on cookies, such as in HTTP, or identifiers, such as in DNS. Trivially, challenge-response mechanisms are ineffective against MitM adversaries, since they can eavesdrop on the challenges and copy their values to responses. However, there is 'hope' that challenge-response defences should suffice to foil attacks by off-path adversaries.

In this work we focus on DNS security against off-path adversaries, especially in light of the recent standardisation efforts to further enhance DNS security with challenge-response mechanisms, [RFC6056, RFC4697, RFC5452]. These recommendations were standardised following Kaminsky's DNS cache poisoning attack in 2008, but most of them were known security measures also before.

In a DNS cache poisoning attack the attacker triggers a DNS request and then sends multiple spoofed responses, each containing different values, trying to guess the correct challenges. The first response with the correct values in challenge fields is accepted and cached by the DNS resolver; subsequent responses are ignored.

The attacker can use DNS cache poisoning to redirect clients to incorrect addresses, e.g., for spam, or malware distribution, credentials theft, or even to gain MitM capabilities for communication to the victim domain.

1.1 Challenge-Response Authentication

Following Kaminsky's cache poisoning attack, DNS resolvers were quickly patched to support challenge-response defences against cache poisoning. Most existing challenge-response mechanisms are 'patches', randomising and validating existing fields in the TCP/IP protocols. We next review standardised and most commonly used challenge-response authentication mechanisms.

DNS uses a random 16-bit *TXID (transaction identifier)* field that associates a DNS response with its corresponding request. DNS implementations additionally support a random selection of name servers each time they send a request. The main defence, that makes poisoning impractical is a (16-bit) *source port randomisation* recommended in [RFC5452], which together with a TXID result in a search space containing 2^{32} possible values; a source port identifies the client-side application in requests, and is echoed (as a destination port) in responses. Specific recommendations for port randomisation algorithms were recently provided in [RFC6056]. Due to the significance of port randomisation for preventing off-path attacks, e.g., cache poisoning and injections into TCP, multiple studies were conducted to measure support of port randomisation in the Internet, and it seems that many resolvers adopted port randomisation methods that were recommended in [RFC6056]; we also confirmed this using CAIDA's data traces [1], see Section 3. Furthermore, a number of DNS checker services, e.g., [3–5], were set up to enable clients to validate predictability of the ports supported by client systems, and algorithms recommended in [RFC6056] are reported secure.

Indeed, security of most DNS resolvers relies on these challenge-response mechanisms, and support of TXID together with source port randomisation, are believed to provide sufficient defence against attacks by off-path adversaries.

Notice however, that port randomisation, as well as other challenge-response mechanisms, do not prevent attacks by MitM adversaries. To protect DNS against a MitM, a cryptographic mechanism, DNSSEC [RFC4033-4035], was standardised already in 1997. DNSSEC is a standard for signing DNS records, allowing resolvers to validate DNS responses. However, so far, DNSSEC is not widely deployed, both at the zones as well as at the resolvers. For example, Google reports that less than 1% of the DNS records it retrieves are signed; and [6, 7] tested queries to `org` and found that 0.8% of the resolvers were validating. Clearly, the deployment of DNSSEC is still very limited.

The goal of our work is twofold. (1) The vulnerabilities that we found and present in this work indicate the dangers of incorrect modelling of adversarial capabilities. In this work we present techniques allowing off-path attackers to efficiently reconstruct the (believed to be secure) ephemeral ports that are allocated in random kernel mode, thus allowing to derandomise standard port randomisation algorithms. Our techniques use fragmented DNS responses in order to elicit timing side channels. These side channels can be used to identify the ephemeral ports' sequence used by the victim system to a specific destination. We recommend fixes against our attacks in the short term. We notified the operating systems vendors and recent Linux kernel was patched to support our (immediate) short term recommendations [8]. (2) Our main message is to emphasise the significance of cryptographic defence, DNSSEC, which provides systematic protection even against other unforeseen vulnerabilities which may be discovered in the future, and prevents attacks not only by off-path but also by a stronger MitM adversary. We hope that the vulnerabilities that we found will encourage wider adoption of DNSSEC.

1.2 Related Work

Recently, a number of DNS cache poisoning attacks were published. We review them and put our contribution in context. The attacks can be grossly categorised into three distinct classes: (1) injection of spoofed records via fragmentation, (2) resolvers behind middleboxes and (3) source port inference.

Injection of Spoofed Records. When a DNS response is fragmented and the second fragment is sufficiently large to contain a DNS record, an attacker can replace the second fragment with a spoofed one, which contains malicious records, e.g., redirecting the client to incorrect hosts, [9]. Most DNS responses are not fragmented, however, [9], presented techniques allowing to cause fragmentation.

Resolvers-Behind-Middleboxes. Resolvers behind middleboxes, e.g., NAT devices or upstream forwarders, is a common setting in the Internet. However, attacks were shown allowing port inferences in both settings, [10–12]. The idea behind attacks is that the middlebox allows to attack the 2^{32} search space sequentially: the attacker first learns the port, and then, when it known the port, it sends 2^{16} packets to match the correct TXID.

Source Port Inference. Kernel processing of incoming packets introduces side channels which can be used to differentiate an open port from a closed one. Recently, [13] showed how to apply it for port inference. However, the techniques are effective on LANs and may be not suitable when the attacker is located on a different network due to the noise introduced by the routers and intermediate Internet devices.

In our work we improve over the results in [13], and present an effective and a much more efficient technique, requiring much less traffic and resilient to network noise.

Our Contributions

We show how off-path attackers can exploit fragmented DNS responses in order to infer ports allocated by common operating systems that support algorithms recommended in [RFC6056]. The ability to predict ports can be used for DNS cache poisoning, and we show how to extend our attacks to poison patched DNS resolvers software; we tested our attacks against Bind 9.8.1 and Unbound 1.4.19 DNS software.

Our techniques improve over the attacks presented in [10], which showed how to predict ports of the resolvers located behind NAT devices. The limitation of [10] is that the attacks apply only when resolvers are behind NAT devices, and require a user-priviledged malware on the LAN. We show that using our techniques much more efficient attacks are possible, which also do not require a malware, and apply to a general scenario, as well as to resolvers behind NAT devices.

We show how to extend our attacks for DNS responses interception. The limitation of applying our techniques for packet interception is that it requires a compromised host on the same LAN with the resolver, behind NAT or firewall device. DNS responses interception allows to circumvent more sophisticated defenses against cache poisoning, such as Eastlake cookies.

2 IP-Defragmentation Cache-Poisoning

In this section we present the basic technique *IP defragmentation cache poisoning* which we apply as a building block throughout the rest of this paper.

IP Fragmentation. TCP/IP networks impose a limit, maximal transmission unit (MTU), on the size of IP packets that they can support. If a packet exceeds the MTU of the link to which it is being forwarded, it is fragmented to smaller packets, i.e., fragments. Each fragment, of the original IP packet, is stamped with the same IP identifier (IP-ID) as the IP-ID value in the original IP packet, and is marked with an offset that corresponds to its location in the byte stream of the original IP packet.

Fragments are stored in an IP defragmentation cache at the destination. When all fragments comprising the original IP packet are received, they are reassembled. Operating systems typically impose a limit on the number of cached fragments per each (source, destination, protocol) triple. For example, in recent versions of the Linux kernel, the default value is 64, and older versions support up to several hundreds of fragments. This limit is imposed via the *ipfrag_ max_ dist* variable; see [14]. In order for fragments to be reassembled, they must match in four parameters: source and destination IP addresses, transport protocol and the IP-ID field. In IPv4, the IP-ID is a 16 bit field[1] selected by the source.

If some fragments are lost or missing, the cached fragments cannot be reassembled and are discarded after a timeout (default value is 30 seconds). The reassembled IP packet is then moved from the defragmentation cache, for transport layer processing.

We performed a study of typical DNS response sizes of Top Level Domains (TLDs) and top Alexa domains, [15], signed and non-signed; the results are plotted in Figure 2. As can be seen, DNS responses signed with DNSSEC result in much larger packets than traditional (unsigned) responses, and even the 'non-existing domain (NXD)' responses that are signed, often exceed the maximal transmission unit (MTU).

[1] In IPv6, the IP-ID is 32 bits; we focus on IPv4, since adoption of IPv6 is still limited.

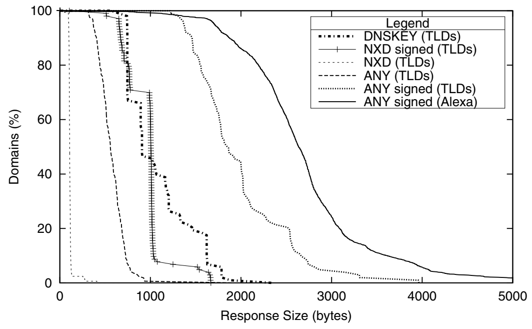

Fig. 2. Length of responses for signed and non-signed Alexa and TLDs, for `ANY`, `DNSKEY` and `A` resource records; `A` records were sent for random subdomains of tested domains, and resulted in NXD responses

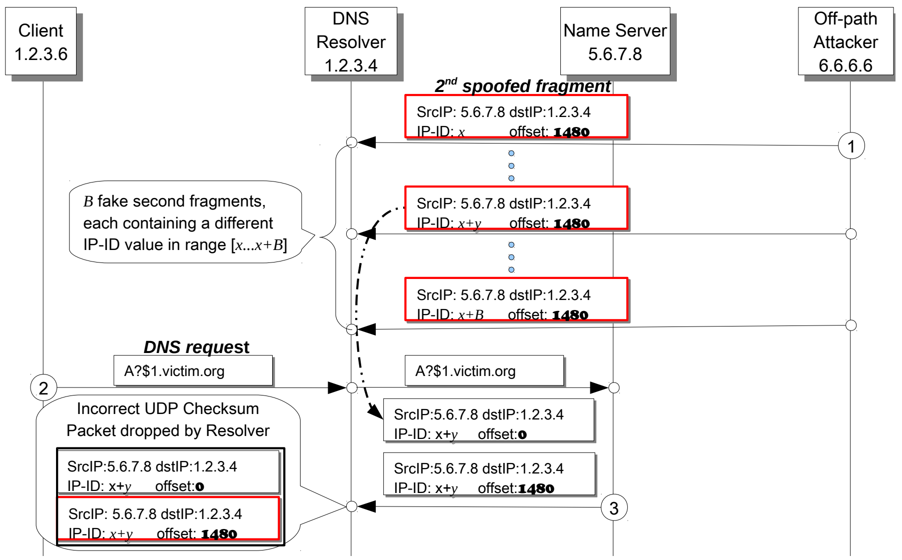
Client 1.2.3.6
DNS Resolver 1.2.3.4
Name Server 5.6.7.8
Off-path Attacker 6.6.6.6
2nd spoofed fragment
SrcIP: 5.6.7.8 dstIP:1.2.3.4 IP-ID: x offset: 1480
SrcIP: 5.6.7.8 dstIP:1.2.3.4 IP-ID: x+y offset: 1480
SrcIP: 5.6.7.8 dstIP:1.2.3.4 IP-ID: x+B offset: 1480
B fake second fragments, each containing a different IP-ID value in range [x...x+B]
1
2
3
DNS request
A?$1.victim.org
A?$1.victim.org
SrcIP:5.6.7.8 dstIP:1.2.3.4 IP-ID: x+y offset:0
SrcIP:5.6.7.8 dstIP:1.2.3.4 IP-ID: x+y offset:1480
Incorrect UDP Checksum Packet dropped by Resolver
SrcIP:5.6.7.8 dstIP:1.2.3.4 IP-ID: x+y offset:0
SrcIP: 5.6.7.8 dstIP:1.2.3.4 IP-ID: x+y offset: 1480

Fig. 3. Defragmentation-cache poisoning via a spoofed second fragment

Replacing IP Fragments. Defragmentation-cache poisoning is reassembly of spoofed IP fragments together with authentic fragments into a correct IP packet. To perform IP defragmentation cache poisoning, the attacker has to cache in the IP defragmentation cache of the victim, spoofed fragments, which contain the same IP addresses as the authentic fragments, sent by the victim, the same IP-ID and protocol fields. Using IP defragmentation cache poisoning, the attacker can replace any authentic fragment, first, middle or last, with a spoofed fragment.

In order for IP to merge a spoofed fragment with a legitimate fragment, the two fragments must match in four parameters: source and destination IP addresses, transport protocol and the *IP identifier (IP-ID)* field. In our setting, the attacker knows the IP addresses, and the transport protocol is UDP. Hence, the only parameter, which the attacker may not know, is the value of the IP-ID. A naive (brute-force) strategy is to try all possible IP-ID values, by sending multiple spoofed fragments, each containing a different IP-ID value. The efficiency of the attack can be significantly improved since most servers support predictable IP-ID values. We collected statistics for name servers of top-level domains (TLDs) and found the following common IP-ID allocation methods: *sequentially incrementing* (supported by more than 70% of the name servers) and *random* (supported by less than 1% of the name servers); see [9], for techniques to predict and hit the correct IP-ID for each allocation method.

Let B denote the number of spoofed fragments sent by the attacker. The defragmentation-cache poisoning attack for a special case[2] where attacker replaces a second authentic fragment with a spoofed one is illustrated in Figure 3. The attack begins when the attacker sends B spoofed second fragments (step 1), which are stored at the defragmentation cache of the destination (for 30 seconds by default), and triggers a DNS request via a puppet (step 2). If one of the B spoofed fragments, that the attacker sent, matches the reassembly parameters in the authentic first fragment, they are reassembled.

The only value that the attacker may need to guess is the IP-ID value in responses from the name server. The probability that the IP-ID of a legitimate (fragmented) response matches the IP-ID of one of the (up to B) spoofed second fragments, which the attacker sent, depends on the IP-ID assignment method; see analysis of the efficiency of defragmentation-cache poisoning for common IP-ID allocation methods: *incrementing* and *random* in [9], we briefly provide it here for completeness.

Random IP-ID. In a random IP-ID allocation the name server selects the IP-ID values in each response uniformly. Let n be the number of DNS requests triggered by the attacker and B the number of spoofed second fragments sent by the attacker. The probability for a successful poisoning can be expressed as follows:

$$\Pr[success] \cong 1 - \left(1 - \frac{B}{2^{16}}\right)^n \qquad (1)$$

See graph representing defragmentation cache-poisoning success probability, based on Eq. (1), in Figure 4. As can be seen, for a default defragmentation buffer of $64 = 2^6$ and 256 parallel DNS requests, the chances are rather slim, i.e., below 0.4. However, random IP-ID is not common among servers, e.g., less than 1% of top-level domain name servers support it, and most deploy (variations of) incrementing IP-ID assignment. As we next show, incrementing IP-ID allows much more efficient prediction strategy.

[2] Extension to a general case is easy, and in this work, we focus on replacing second fragments.

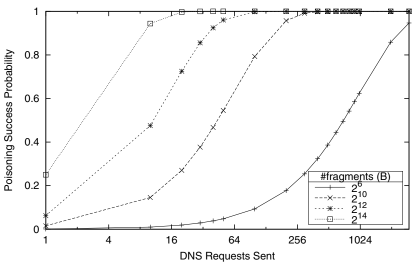

Fig. 4. Defragmentation-cache poisoning success probability per attempt, by analysis (Eq. (1)), for $B \in \{64, 1024, 4096, 16384\}$ (number of fake second fragments in cache)

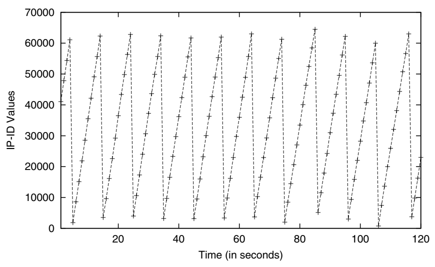

Fig. 5. Progress of globally incrementing IP-ID values of `a0.org.affilias-nst.info` name server of `org` TLD.

Incrementing IP-ID. A significant fraction (more than 70%) of the name servers use incrementing IP-ID allocation methods. An attacker can query the name server directly and find out the current IP-ID value (since the same counter is used for sending packets to the attacker and to all other destinations, including the resolver). However, the IP-ID may considerably change between the query of the attacker and the query of the resolver, since the name server receives queries from other sources too. Therefore, in busy nameservers, that receive queries at a high rate, the IP-ID may grow very rapidly. Notice though that even if the DNS requests' rate to popular name servers is high, it is typically predictable. For example, in Figure 5, we show the measurements we ran on `a0.org.affilias-nst.info`, one of the name servers of `org`, that supports globally-incrementing IP-ID allocation; notice how rapidly the IP-ID 'grows' across the cyclic 16-bit counter field, yet it can be seen that the increments are predictable.

Indeed, the IP-ID value can be extrapolated, and the reason for this is that the query rate to name servers is stable, see [16].

To find the IP-ID value, the attacker measures the rate at which the name server receives requests, then measures the latency between itself and the victim resolver, and estimates the latency between the victim resolver and the name server. Then it samples the current IP-ID value, by seding a query to the name server. The attacker uses the response from the name server to extrapolate the value of the IP-ID that will be assigned by the name server to the response that it will generate for the resolver.

3 Port Derandomisation via IP Defrag-Cache Poisoning

In this section we describe port randomisation methods, standardised and deployed in popular systems, and show how to apply IP defragmentation cache poisoning, described in Section 2, for port derandomisation and discovery. IP defragmentation cache poisoning was applied in prior art, but mainly for attacks on performance, e.g., to ruin an IP packet, e.g., for denial of service attacks see [RFC6274] and [17], or name server pinning, [10] or for injection, [9]. In this work we present the first port derandomisation attacks using defragmentation cache poisoning. Our techniques apply to standard, and widely deployed, port assignment methods, supported by popular operating systems. This allows off-path attackers to predict client ports for a wide range of attacks, including DNS cache poisoning.

The Myth of Per-Destination Ports Security. A globally incrementing port allocation is not considered secure, since the attacker can sample the current port value, and then use it to extrapolate the next port value that will have been assigned by the client to its DNS requests. Indeed, most systems currently support a per-destination incrementing port allocation algorithms, recommended in [RFC6056]. A per-destination incrementing port is believed to be secure, since different ports' sequences are assigned by the resolver to different destinations; in particular learning the port value to one destination does not leak the port value assigned to some other destination. We checked, in [11], the *predictability rate* assigned by the popular DNS checker service provided by the OARC [4], to resolvers that send DNS requests with per-destination incrementing port. The tool reported (the highest) GREAT score, indicating that per-destination allocation methods are believed to be secure by the DNS experts. However, our results (within) show otherwise.

The idea behind per-destination incrementing ports is that a different sequence is selected to each destination as follows: the first port to some destination is selected at random, and subsequent packets to that destination are assigned sequentially incrementing ports. As a result, each destination knows only the port sequence that is used for communication to it, and cannot learn anything about communication to other destinations. Per-destination ports allocation underlies most of the algorithms proposed in [RFC6056]; more details in Section 3.2.

We collected statistics, [11], from two CAIDA datasets from 2012 [1] and found that many DNS requests support incrementing ports; the packets' traces are collected by CAIDA on (several) *backbone (OC192) links.* We used the traces to collect all the DNS requests (destination port 53) over UDP, and then filtered out IP addresses with a single DNS request, and collected only the sources that sent two or more requests. This allowed us to infer information about the source port allocation of the remaining DNS requests. We found that 54% of the requests were sent from incrementing ports.

In this section, we present attacks against standard and widely deployed port randomisation algorithms. We first show an attack against a popular per-destination algorithm, supported by Linux kernel OS, and then show extensions against other algorithms recommended in [RFC6056].

3.1 Predicting Linux OS Ports

In Linux OS kernel, the initial port to some destination is selected at random, and subsequent ports to that destination are incremented sequentially. This method is in fact a special case of Algorithm #3 [RFC6056] in Section 3.2, except that a distinct counter is maintained to each destination (instead of a global counter for all destinations). We show how to apply fragmentation to traverse the ports' range, in descending order, from the highest to lowest until correct port is found[3].

When traversing the ports' range, at each iteration i the attacker can sample more than a single port. For instance, if attacker samples p ports at each iteration, in the worst case, the attacker 'meets the resolver' after $\frac{2^{16}}{2^p}$ attempts.

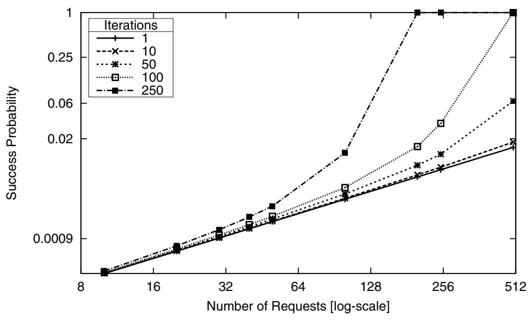

Fig. 6. Discovery of 'random' ephemeral ports' sequences against Linux Kernel, by applying defragmentation cache poisoning. The attacker uses the latency of DNS responses as a timing side channel to learn whether the correct port was hit.

The search distribution is composed of ports' pool and of the TXIDs range. We next show how to apply IP defragmentation-cache poisoning to *split* the distribution of TXIDs and ports values to two separate distributions of size (at most) 2^{16} each (assuming an ideal case where a maximal number of ports is used).

The steps of the attack are illustrated in Figure 7. We assume that the attacker already discovered the IP-ID value using techniques described in Section 2.

The idea is to use fragmentation to overwrite the transport layer header of the fragmented IP packet sent by the name server to the resolver. In each such attempt the attacker sends a spoofed fragment with a source IP of the name server and includes a guess for a port. If the guess is correct - the response is accepted and cached by the resolver. Otherwise, if the port in the spoofed fragment is incorrect - the resolver rejects the response, and retransmits the

[3] Often not all ports are used by the OS. DNS running on Windows server 2008 uses ports range (49152−65535) and Windows 2000/XP/server 2003 use ports from range (1025 – 5000). Older Bind versions use fixed ports. This results in ranges that are significantly smaller than 2^{16}, e.g., it is considered safe to use ports in the range (1024 – 49152), [RFC5452].

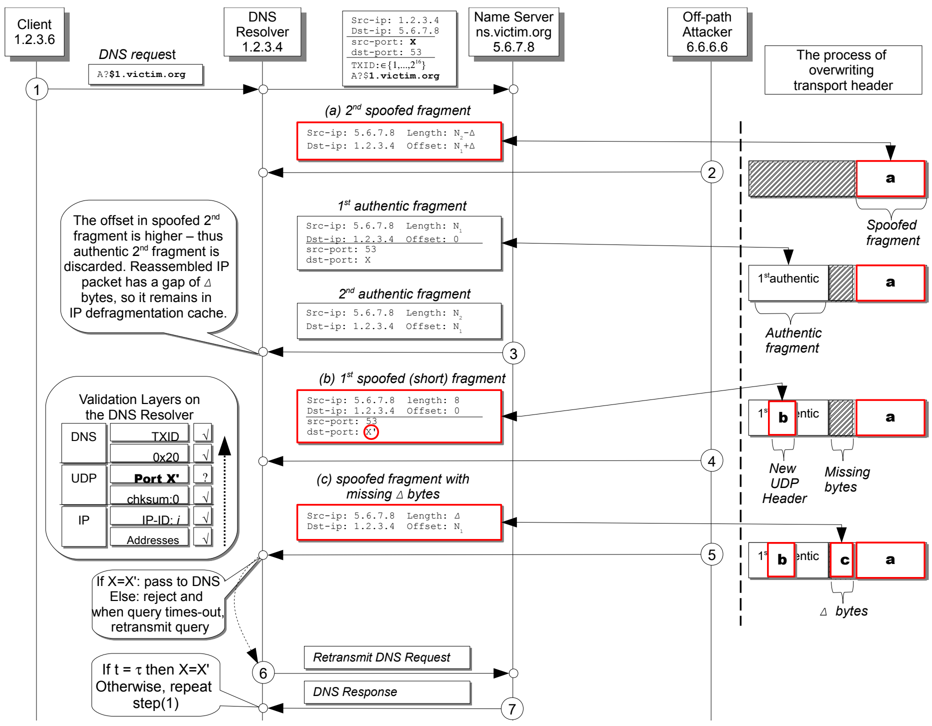
Client 1.2.3.6
DNS Resolver 1.2.3.4
Name Server ns.victim.org 5.6.7.8
Off-path Attacker 6.6.6.6
The process of overwriting transport header
DNS request
A?$1.victim.org
Src-ip: 1.2.3.4 Dst-ip: 5.6.7.8 src-port: X dst-port: 53 TXID:∈{1,...,2^16} A?$1.victim.org
(a) 2nd spoofed fragment
Src-ip: 5.6.7.8 Length: N2-Δ Dst-ip: 1.2.3.4 Offset: N1+Δ
1st authentic fragment
Src-ip: 5.6.7.8 Length: N1 Dst-ip: 1.2.3.4 Offset: 0 src-port: 53 dst-port: X
2nd authentic fragment
Src-ip: 5.6.7.8 Length: N2 Dst-ip: 1.2.3.4 Offset: N1
The offset in spoofed 2nd fragment is higher – thus authentic 2nd fragment is discarded. Reassembled IP packet has a gap of Δ bytes, so it remains in IP defragmentation cache.
(b) 1st spoofed (short) fragment
Src-ip: 5.6.7.8 length: 8 Dst-ip: 1.2.3.4 Offset: 0 src-port: 53 dst-port: X'
Validation Layers on the DNS Resolver
DNS TXID √ 0x20 √
UDP Port X' ? chksum:0 √
IP IP-ID: i √ Addresses √
(c) spoofed fragment with missing Δ bytes
Src-ip: 5.6.7.8 Length: Δ Dst-ip: 1.2.3.4 Offset: N1
If X=X': pass to DNS Else: reject and when query times-out, retransmit query
If t = τ then X=X' Otherwise, repeat step(1)
Retransmit DNS Request
DNS Response
1 2 3 4 5 6 7
a
Spoofed fragment
1st authentic
Authentic fragment
b
New UDP Header
Missing bytes
c
Δ bytes

Fig. 7. DNS request port discovery: in step 1, the attacker plants a spoofed fragment in the resolver's defragmentation cache and then in step 2 the puppet triggers a DNS request to a resource within the victim's domain. The off-path attacker, in step 3, sends its guess for a port. If failure, timeout event occurs, and the resolver retransmits the DNS request. Otherwise, the attacker guessed the correct port.

request. The attacker uses the time till the response arrives, as a side channel, to distinguish between the correct and an incorrect ports.

In order to overwrite the (UDP) transport layer header, the attacker has to craft a spoofed first fragment, which is smaller than the original first fragment. The spoofed first fragment is 8 bytes long, and contains a guess for a new port; its purpose is to overwrite only the UDP header (also 8 bytes long) in the authentic first fragment. Notice that when two fragments contain identical offsets, then the last arriving fragment overwrites the first. Therefore, in order for the spoofed fragment to overwrite the transport header of the authentic fragment, it must arrive at the resolver *after* the first authentic fragment, and *before* the IP packet is reassembled, when the authentic second fragment arrives. Notice that this is not trivial, since once the two legitimate fragments arrive, they are immediately reassembled. Thus the attacker has to ensure that the first legitimate fragment remains in the defragmentation so that its header can be overwritten. To achieve that, the attacker has to plant a spoofed second fragment, similarly to attack in Section 2, which will be reassembled with the legitimate first fragment.

Attack. We next describe the steps of the attack.

Let $f = f_1||f_2$ be the IP packet consisting of two fragments f_1 starts at offset 0 and is of length N_1 and f_2 starts at offset N_1 and is of length N_2.

(1) attacker triggers a DNS request (whose response is fragmented).

(2) the attacker sends a spoofed second fragment f_2', starting at offset $(N_1+\Delta)$, where Δ is some number of bytes.

(3) When the first authentic fragment f_1 arrives it is reassembled with the spoofed second fragment f_2' that is already waiting in the defragmentation cache; when the authentic second fragment f_2 arrives, it is discarded since a spoofed fragment starts and ends at a higher offset $(N_1 + \Delta)$. However, the reassembled IP packet does not leave the defragmentation cache since there is a gap of Δ bytes that are still missing.

(4) The attacker sends a short fragment that overwrites *only* the UDP header in the original first fragment. This fragment overlaps with the first 8 bytes (the UDP header) in the authentic first fragment; the spoofed fragment contains `checksum` 0, which indicates that checksum validation is disabled[4], more fragments is set to 1 (`mf`=1), and `offset` is 0. When initiating the attack, the attacker sets the UDP port in this spoofed first fragment to 2^{16}, and decrements its value during each subsequent iteration.

(5) Finally, the attacker sends a fragment that starts at an offset N_1 and is of size Δ. This fragment fills the missing gap.

Following reassembly, these fragments result in a complete IP packet, that leaves the IP defragmentation cache and is passed to upper UDP layer.

If the attacker guessed the port correctly, in step (4), the DNS resolver will cache and forward the response to the puppet. In contrast, if the port is incorrect, the resolver will discard the DNS response; as a result, a timeout event will occur, and the resolver will retransmit the request again. The attacker uses the differences in responses' latencies as a side channel to detect guessing the correct port.

Analysis. Let r be a DNS response size in bytes. Let R bytes/sec be the transmission rate of the attacker. Let t seconds be a limit on the timeout for a DNS request (i.e., including all retransmitted requests for that query) and let q be a number of times a pending query is retransmitted until it is terminated and SERVERFAIL is returned.

Resolvers implement retransmission policy based on round trip time estimates of the name servers, [RFC1536], and support timeout management with exponential backoff. When a timeout occurs resolver enters an exponential backoff phase, i.e., the timeout is doubled, and query is retransmitted. Resolvers implement variable timeout and retransmission values, typically up to 45 seconds (which is also a recommended ceiling for total timeout for a query [RFC1536]), and attempt up to 15 retransmissions. For instance, Unbound 1.4.19 sets t to a

[4] UDP checksum validation is optional, and it can be disabled by name servers by setting it to 0 (0000 in hexadecimal). When the checksum is disabled it is not validated by the resolvers.

maximal value of 40 seconds and Bind 9.8.1 sets $t \leq 30$ seconds and $q \leq 10$, i.e., supports up to 10 retransmissions before terminating a query.

In each retransmission the resolver advances the port (in case an incrementing allocation is supported). This allows the attacker to sample a number of ports in a single iteration (since with each retransmission there is a new pending request). Once the port is known the attacker launches a DNS cache poisoning attack, i.e., sends 2^{16} spoofed DNS responses, for some victim domain, such that each response contains a different TXID value.

The number of iterations required to hit the correct port in the worst case is: $\frac{2^{15}}{(q+1)}$ for a per-destination incrementing port assignment; during each iteration the attacker matches the original query and up to q query retransmissions. Since the attacker does not need to match the TXID, at each iteration only 3 fragments are sent; this significantly reduces the complexity of the naive cache poisoning attack, where the attacker attempts to hit both the correct port and the correct TXID.

The worst-case number of requests required to guess the port is $\frac{2^{15}}{(q+1)}$. During each iteration $3(q+1)$ fragments are sent, thus the worst case number of fragments is $3(q+1) \cdot \frac{2^{15}}{(q+1)} = 3 \cdot 2^{15}$.

Once the port sequence is known, the attacker launches the traditional DNS cache poisoning attack: (1) triggers a DNS request and (2) generates and transmits 2^{16} forged DNS responses (with a spoofed IP address of the victim name server) such that each DNS response contains a different value of the TXID, and the port number, which was guessed earlier. One of the responses, that contains the correct value of TXID, is accepted and cached by the resolver. The analysis of the attack is presented in Figure 6.

We next calculate the success probability, the required worst case number of attack iterations and the number of requests triggered by the puppet, and the number of fragments sent by an off-path attacker.

The probability of hitting the correct port in a single attempt, when triggering Q DNS queries, is: $\Pr[success] = \frac{Q}{2^{16}}$.
The success probability during i^{th} iteration of the attack is:
$\Pr[success] = \frac{Q}{2^{16} - i \cdot Q}$.

In the worst case the attack has to be repeated at most $i \leq \frac{2^{16}}{2 \cdot Q} = \frac{2^{15}}{Q}$ iterations (assuming that at each iteration the attacker sends Q queries). The number of DNS requests, sent by the client, in the worst case is: $Q \cdot \frac{2^{15}}{Q} = 2^{15}$. Total number of sets of fragments (i.e., three fragments each time, as described in Figure 7), sent by off-path attacker is: $N \cdot \frac{2^{15}}{Q}$.

Notice that the attacker can reduce the number of iterations by sampling more ports in each iteration, and it can also reduce the number of requests in each iteration by circumventing a 'birthday protection'.

3.2 Inferring Ports Supported by other Algorithms

In each following subsection, we give an abstract presentation of the port allocation method recommended in [RFC6056] and show how it can be circumvented using IP defragmentation cache poisoning.

Notice that only Algorithms #1 and #2, [RFC6056], select random port for each outgoing packet, and thus are not vulnerable to port prediction attacks via fragmentation. However, they are vulnerable to port exhaustion attacks, [10], whereby the attacker occupies all the available ports except one, which is then assigned to the query of the resolver.

Simple Hash-Based Port Selection. Simple hash based port assignment method is described in Algorithm #3, [RFC6056]. The port selection algorithm, at the sender, maintains a *different offset* to each destination. The offset is a result of a pseudorandom function computed over a tuple (*source IP address, destination IP address, destination port, secret key*). The algorithm uses *a single counter*, incremented globally, by one, for each port allocation, and added to the offset of the relevant destination. As a result, connections to different destinations will have different sequences of port numbers, and one destination should not be able to anticipate the ports allocated to the other. This algorithm was supported by Linux OS kernel version 2.6, and was believed to be secure against off-path attackers. Recent Linux versions also use a variation of per-destination incrementing ports, see Section 3.1.

We show how to apply our IP defragmentation cache poisoning technique to discover the ephemeral port used by a victim resolver to some destination name server. The attacker learns the typical delay δ for a request to some name server. The attacker then triggers DNS requests (via the puppet on the LAN), and sends spoofed fragments for each candidate port that it samples, starting with the highest port, and decrementing the port during each attempt. The attacker learns when the correct port is hit, via the response latency. The ephemeral port is found if during sampling of a candidate port z the response arrives after $\tau > \delta$ seconds. Otherwise, the attacker repeats the attack with a new port $z-1$; following a descending order of ports. When the correct port is hit, the resolver accepts the DNS response. Otherwise, when an incorrect port is hit, the resolver discards the response. This results in query retransmission by the resolver, and adds more than a second to the response[5].

Once the port is discovered, the attacker can use its value for attacks at some later time, without the need to discover the port again. This is due to the fact that the same counter is used to all destinations, thus the attacker only needs to trigger a DNS request to a server that it controls, to check the current counter value, and adjust the port value accordingly.

Double-Hash Port Selection. Algorithm #4, [RFC6056], builds on Algorithm #3 (Section 3.2), but instead of keeping a globally incrementing counter

[5] Typical latency for a DNS request and response is in the order of 100 ms.

for all destinations, it groups the destinations to m sets and uses a separate counter to each set. The recommended value of m is 10, [18,19], however, as [19] notes, larger m values provide for better port obfuscation, since the same counter is not shared between too many clients. Notice that the special case of $m = 1$ is supported in Linux kernel (Section 3.1).

Port derandomisation attack proceeds as follows: first, the attacker attempts to find an IP address that *falls within the same set as the target name server* (the goal of the attacker is to eventually discover the port, used by the resolver, to that name server); the attacker may employ for its attack a number of compromised hosts, e.g., bot computers that it controls (located in different networks and not necessarily on the same LAN with the resolver). The idea is to trigger a request to some host i, and then launch IP defragmentation cache poisoning to check if the current port, allocated to the target server, was incremented; the port sampling attack steps are similar to those presented against *simple hash-based port selection.* If the port was incremented, then the host at IP address i falls within the same set as the target name server. This host i can be used to sample port increments.

Random-Increments Port Selection. Algorithm #5, [RFC6056], maintains a globally incrementing counter to all the destinations, which is incremented by (a randomly selected) N at each invocation of the ephemeral port allocation procedure. For $N = 1$ this is exactly the globally incrementing algorithm implemented in Windows and FreeBSD operating systems.

The attack against *simple hash-based port selection* applies with a slight modification: the attacker has to sample N ports, instead of one, in each attempt. Notice, that the actual value selected by the port allocation procedure can be less than N, but since the attacker does not know its value, it has to sample N ports each time in the worst case. Furthermore, when hitting the correct port, the attacker cannot tell which, out of the sampled N ports, it was, and it only learns that it was one the sampled ports. To reduce the success of port derandomisation attacks, N should be as large as possible. On the other hand, large N increases the chance for port collisions due to ports' reuse from previous connections, and smaller values of N are required to reduce collisions.

4 Resolvers behind NAT Devices

A natural question is what is the impact of port derandomisation attack against resolvers behind NAT devices. In fact, the same attack step, illustrated in Figure 7, apply, since the NAT itself reassembles IP fragments in order to be able to forward the packet to the correct internal host on the LAN based on the ports. This also improves over the attack in [10] which required a zombie order to leak the port number used by the NAT, in packets payload, to the external attacker, and used significantly more packets, rendering the attack impractical.

4.1 Are They Common?

To estimate what fraction of resolvers are located behind NAT devices, we ran statistics on two CAIDA datasets from 2012 [1], that were collected on equinix-chicago and equinix-sanjose monitors on high-speed Internet backbone links. Both traces contained packets sent from distinct 89750 source IP addresses, collected over two minutes interval. We ran the following test to check for DNS resolvers behind NAT devices: (1) we collected all DNS requests, i.e., packets sent to port 53; (2) we created a set of IP addresses that sent at least one DNS request per second (to ensure that we do not mistakenly interpret a host for a resolver behind a NAT; (3) we then parsed the traces to check if those IP addresses also sent packets to other ports, including port 80, and 443. We then concluded with high probability that those resolvers were behind NAT devices. We came up with a total of 3492, out of 89750, resolvers behind NAT devices.

4.2 Packet Interception

In a setting where a victim resolver is located behind a NAT device, and the off-path attacker controls a host on the same LAN, it can use the technique described in Section 3 to *intercept* DNS requests sent by the resolver. Notice that such a setting is common, e.g., the attacker may be a legitimate user on the same organisational network with the resolver, or on a public wireless access network, or it may control an infected host, i.e., user space malware would suffice (many computers are infected with malware), on the same network with a resolver. The same attack steps as in Section 3 apply, except that the target is the NAT device. The internal host first sends a packet (or a number of packets) to the name server, in order to create a mapping in the NAT table. The off-path attacker sends the spoofed fragments (as in Section 3), to overwrite the port of authentic response from the name server. The outcome of overwriting the port of inbound packets traversing the NAT, is that the DNS response is sent to a different internal host, i.e., the one that is identified by the new port which was used by the off-path attacker in its spoofed fragment, instead of the designated recipient (i.e., the resolver). Such attacks would enable circumvention of Eastlake cookies, and all other *non cryptographic defences.*

5 Defenses

The vulnerabilities described in this paper are severe and apply to many systems that support (variations of) per-destination ports assignment. We recommend a number of short term client-side defences. Most notably, proper port randomisation, which would prevent these attacks. Client side defences also include firewall-based mechanisms, such as [20]. In particular, in [20] we showed two techniques which make cache poisoning impractical: (1) we showed how to artificially increase the number of IP addresses allocated to a resolver, and (2) how to detect poisoning attacks by keeping track of responses which contain incorrect challenge-response values.

Another simple defence is to deploy IPv6. Its much larger IPv6 address space, makes the attack impractical.

In the long term we recommend to deploy systematic defences, most notably DNSSEC [RFC4033-RFC4035], which is the best defense against the poisoning attacks. DNSSEC provides security not only against off-path but also against MitM attackers. However, deployment faces multiple challenges and obstacles, see [7, 21, 22] for details. We hope that our results will help motivate speeding up adoption of DNSSEC and focusing efforts on investigation of the deployment challenges.

6 Conclusions

We presented port inference techniques using timing side channels which IP defragmentation cache poisoning makes available. Our techniques allow to predict source ports selected by client systems supporting port randomisation algorithms recommended in [RFC6056]. Our attacks effectively circumvent the 'source port randomisation (SPR)' defence making DNS cache poisoning, and other attacks which rely on ports' prediction, practical. We applied our techniques for DNS cache poisoning attacks agaist standard DNS resolvers, Bind 9.8.1 and Unbound 1.4.19, and popular operating system, Linux kernel OS. This attack is significant, since a large and growing number of networks support per-destination algorithms recommended in [RFC6056].

Acknowledgements. We are grateful for support for CAIDA's Internet Traces [1] that is provided by the National Science Foundation, the US Department of Homeland Security, and CAIDA Members.

References

1. Paulo, S.L.M., Barreto, S.D., Galbraith, C.O.: hEigeartaigh, and Michael Scott. Efficient pairing computation on supersingular abelian varieties. IACR Cryptology ePrint Archive, 375 (2004)
2. Beverly, R., Koga, R., Claffy, K.: Initial Longitudinal Analysis of IP Source Spoofing Capability on the Internet. Internet Society Article (July 2013)
3. Corporation, G.R.: DNS Nameserver Spoofability Test (2012), `https://www.grc.com/dns/dns.htm`
4. DNS-OARC: Domain Name System Operations Analysis and Research Center (2008), `https://www.dns-oarc.net/oarc/services/porttest`
5. Provos, N.: DNS Testing Image (July 2008), `http://www.provos.org/index.php?/archives/43-DNS-Testing-Image.html`
6. Gudmundsson, O., Crocker, S.D.: Observing DNSSEC Validation in the Wild. In: SATIN (March 2011)
7. Lian, W., Rescorla, E., Shacham, H., Savage, S.: Measuring the practical impact of dnssec deployment. In: Proceedings of USENIX Security (2013)
8. Neira, P.: Patchset of Netfilter Updates (2013), `http://patchwork.ozlabs.org/patch/307041/`

9. Herzberg, A., Shulman, H.: Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org. In: The Conference on Communications and Network Security, CNS 2013. IEEE (2013)
10. Herzberg, A., Shulman, H.: Security of Patched DNS. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 271–288. Springer, Heidelberg (2012)
11. Herzberg, A., Shulman, H.: Vulnerable Delegation of DNS Resolution. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 219–236. Springer, Heidelberg (2013)
12. Gilad, Y., Herzberg, A., Shulman, H.: Off-Path Hacking: The Illusion of Challenge-Response Authentication. IEEE Security & Privacy (2014)
13. Herzberg, A., Shulman, H.: Socket Overloading for Fun and Cache Poisoning. In: ACM Annual Computer Security Applications Conference (ACM ACSAC) (December 2013)
14. Kernel.org: Linux Kernel Documentation (2011), `http://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt`
15. Alexa: The web information company, `http://www.alexa.com/`
16. Wessels, D., Fomenkov, M.: Wow, thats a lot of packets. In: Proceedings of Passive and Active Measurement Workshop, PAM (2003)
17. Gont, F.: Security Implications of Predictable Fragment Identification Values. Internet-Draft of the IETF IPv6 maintenance Working Group (6man) (March 2012) (expires September 30, 2012)
18. Larsen, M., Gont, F.: Recommendations for Transport-Protocol Port Randomization. RFC 6056 (Best Current Practice) (January 2011)
19. Allman, M.: Comments on selecting ephemeral ports. ACM SIGCOMM Computer Communication Review 39(2), 13–19 (2009)
20. Herzberg, A., Shulman, H.: Unilateral antidotes to DNS poisoning. In: Rajarajan, M., Piper, F., Wang, H., Kesidis, G. (eds.) SecureComm 2011. LNICST, vol. 96, pp. 319–336. Springer, Heidelberg (2012)
21. Herzberg, A., Shulman, H.: Dnssec: Security and availability challenges. In: 2013 IEEE Conference on Communications and Network Security (CNS), pp. 365–366. IEEE (2013)
22. Herzberg, A., Shulman, H.: Retrofitting Security into Network Protocols: The Case of DNSSEC. IEEE Internet Computing 18(1), 66–71 (2014)