DNSpooq

DNSpooq

Cache Poisoning and RCE in Popular DNS Forwarder dnsmasq

JSOF Research Lab

Moshe Kol

Shlomi Oberman

JSOF

JANUARY 2021

DNSpooq: Cache Poisoning and RCE in Popular DNS Forwarder `dnsmasq`

The JSOF Research Lab

Abstract

In this paper, we report several security vulnerabilities in `dnsmasq` [10], a popular lightweight caching DNS and DHCP server, developed and maintained by Simon Kelley. `Dnsmasq` is a commonly used software component in networking devices, such as routers and access points, and is installed by default in several major Linux distributions. This research has implications for `dnsmasq` in particular, as well as a reference point for security requirements of DNS forwarders in general.

We found that `dnsmasq` is vulnerable to a DNS cache poisoning attack by an off-path attacker (i.e., an attacker that does not observe the communication between the DNS forwarder and the DNS server). Our attack allows for poisoning of multiple domain names at once, and is a result of several vulnerabilities found. The attack can be completed successfully in seconds or a few minutes, and has no special requirements. We also found that many instances of `dnsmasq` are mis-configured to listen on the WAN interface, making the attack possible directly from the Internet. Our estimation is that currently there are about 1M vulnerable `dnsmasq` instances open to the Internet and many more that are not open to the Internet.

In addition, we found a high-severity heap-based buffer overflow vulnerability that could potentially lead to remote code execution when `dnsmasq` is configured to use DNSSEC. The vulnerability resides in the early stages of DNSSEC validation, rendering DNSSEC's defense against DNS attacks ineffective. We have not attempted full exploitation of this vulnerability. Other vulnerabilities we found are heap-based buffer overflows that can only lead to denial-of-service when DNSSEC is used.

We named the collection of vulnerabilities **DNSpooq** (pronounced 'DNS spook'), because the vulnerabilities involve spoofing a DNS cache record. The last letter was changed from 'k' to 'q' to connect it with the name of the affected software – `dnsmasq`.

1 Introduction

DNS cache poisoning is a classic attack on the DNS infrastructure. If successfully executed, it allows an attacker to place a malicious entry in the DNS server cache, redirecting network traffic to an attacker-controlled server. For example, an attacker can place a record stating that `www.bank.com` resolves to an attacker-controlled IP address `6.6.6.6`. Placing such a malicious cache entry affects all clients which resolve `www.bank.com` against the affected DNS server/forwarder.

A general case cache poisoning attack for an off-path attacker operates as follows:

1. Client queries a DNS server for `www.bank.com`.
2. If `www.bank.com` isn't present on the DNS server cache, the server will query another (upstream) DNS server to resolve that name.
3. While the server waits for the response, an off-path attacker sends many malicious responses, appearing to come from the true upstream server.
4. If the attacker manage to correctly guess the TXID and source port of the request, a malicious entry will be inserted into the cache.

The *TXID (transaction ID)* is a 16-bit field present in the DNS header, used to match an incoming response to its corresponding request. Since the server will reject any response which does not match a corresponding TXID, the attacker has to guess the TXID in order for her response to be considered.

In the past, the 16-bit TXID field was the only defense against off-path attackers. In 2008, security research Dan Kaminsky showed [9] that 16 bits of entropy is not enough to protect against cache poisoning attacks. One mitigation that was proposed and widely adopted is to randomize the source UDP port is order to increase the number of bits the attacker has to guess. As ports are also 16-bit, with this approach we reach 32 bits of entropy which is much better. Another mitigation proposed was the 0x20 encoding [2], with which the server chooses the capitalization of the domain name randomly to further increase entropy.

There is no shortage of prior work on DNS security. However, little has been written publicly about the security of `dnsmasq` and DNS forwarders. In 2017, security researchers from Google published [3] several high-severity vulnerabilities affecting several parts of the software, including the DNS resolver. Recently, a group of researchers demonstrated [17] a successful "defragmentation attack" on `dnsmasq`. Though elegant, the attack assumes predictable IPID of packets sent from the recursive resolver, a condition which is easy to centrally fix and is not the case for all recursive resolvers, including popular ones, such as Google Public DNS (8.8.8.8) and Verisign (64.6.64.6). In addition, for the IPID prediction to succeed, an attacker must control a machine on the LAN. This and other limitations makes the defragmentation attack difficult to execute over `dnsmasq` instances open to the Internet. A newer attack, called SAD DNS [11], successfully defeats source-port randomization using a side channel which leverages ICMP "port unreachable" packets together with the ICMP global rate limiting feature. This attack allows for DNS cache poisoning, and is inherently a flaw in the operating system rather than any DNS software.

Our contribution We are reporting 7 vulnerabilities affecting `dnsmasq` (CVE-2020-25681-CVE-2020-25687). Of these, 3 vulnerabilities enable cache poisoning attack, while the other 4 are buffer overflow vulnerabilities, of which one has the potential to take-over the device.

The cache poisoning vulnerabilities enable an attack that exploits a weakness in the software itself, works with the default configuration and does not require special conditions to be met (e.g. IPID prediction). Consequently, it applies to a wide range of scenarios: If the `dnsmasq` instance is open to the Internet, the attack can be executed directly from the Internet. In case `dnsmasq` is not open to the Internet, the attack can also work a user in the LAN simply browses to an attacker-controlled website (see § 4 for details). The attack is also exploitable if the attacker has control over a device on the LAN.

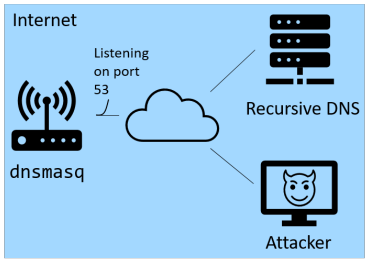
The buffer overflow vulnerabilities are high-severity vulnerabilities that we found and reported in the DNSSEC validation procedure of `dnsmasq`, making the main defense against cache poisoning attacks ineffective. These show, *once again*, that an implementation bug can circumvent the entire protocol security. The vulnerabilities can be combined with the cache poisoning attack to potentially take over a device running a vulnerable `dnsmasq` instance which uses DNSSEC.

2 Overview

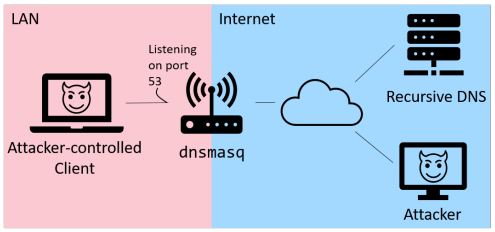
Generally speaking, *DNS forwarders* [6] sit between stub resolvers (e.g. PCs, mobile clients) and recursive resolvers. Their role is to send a received query to the recursive resolver (upstream server), cache the response and relay it to the querying client. Forwarders provide caching facilities to better serve their clients, ultimately reduce latency and bandwidth. They are commonly deployed in networking equipment (such as routers and access points). `Dnsmasq` is one of the most popular caching DNS forwarders, making it an interesting and valuable target for attackers. The uses of `dnsmasq` are varied, and range beyond use only as a DNS caching server. It is used for other purposes such as traffic redirection for ad-blocking or guest network browsing.

The cache poisoning attack affecting `dnsmasq` can be performed in a short time (between seconds to few minutes) with no special hardware requirements and can poison 9 domain names at a time. The attack is a result of different vulnerabilities, described in § 3, and can be accomplished using several methods.

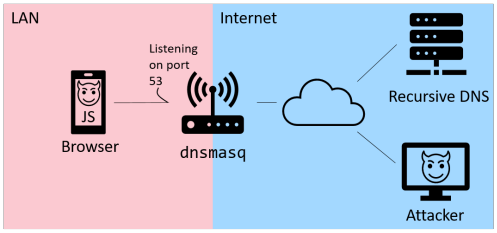
Below we outline several scenarios that can be used to mount the attack on a vulnerable `dnsmasq` instance. For some devices and configurations other attack scenarios may be possible.



(a) Open to the Internet



(b) `dnsmasq` in LAN with attacker-controlled machine



(c) `dnsmasq` in LAN from browser

Figure 1: Cache poisoning attack scenarios

2.1 Scenario 1: Open `dnsmasq` instances

In this scenario, shown in Figure 1a, an attacker can attack `dnsmasq` resolvers listening on port 53 on the WAN interface (open to the internet). As of Sep '20, according to Shodan.io, there are approximately[1] 1M vulnerable `dnsmasq` instances.

To successfully mount this attack, an attacker will need a server on the Internet with the ability to send packets with spoofed source IP and a registered DNS domain name. We have demonstrated that both are easy to acquire.

2.2 Scenario 2: `dnsmasq` in LAN with attacker-controlled machine

In this scenario, depicted in Figure 1b, an attacker can attack a `dnsmasq` resolver that only listens on port 53 on the local LAN. In this case, it is possible to perform the attack completely within the LAN, impacting all devices that are configured to use `dnsmasq` as their DNS server.

To perform the attack, an attacker will need a registered DNS domain name as well as one of the following:

- Control of single device in the LAN network that can spoof source IP packets.
- Control of single device in the LAN network and a server on the Internet with the ability to spoof packets.

Often in the case of a hot-spot, captive portal, or guest network, the client may be considered on the LAN.

So for instance, a malicious user on public/shared machine in an hotel which uses `dnsmasq` as DNS server can affect all the guests of the hotel. The same applies to malicious airport guest using the public captive portal WiFi. Her device is considered in the LAN and a successful execution of the attack can affect all airport network users.

2.3 Scenario 3: `dnsmasq` in LAN from browser

Similar to the previous scenario, here we assume that a `dnsmasq` resolver only listens on port 53 on the local LAN. To perform the attack, an attacker will need a server on the Internet with the ability to spoof packets and a registered DNS domain name, as well as one of the following:

[1] This ranges from 850K-1.2M from day to day. We do not know why.

- A machine in the LAN that browses to a website which executes an attacker-controlled JavaScript on a Safari browser, for example due to being served with a malicious advertisement or clicking on a link.
- Multiple machines browsing with Google Chrome to attacker-controlled content.

Behavior with Edge/IE browsers was not determined.

2.4 Impact

The vulnerabilities result in fake DNS records being injected in to the `dnsmasq` cache for potentially long periods of time. This can lead to several main outcomes:

MITM MITM or read capabilities on different types of traffic like emails, web browsing, captive portal, SIP, SSH, NTP, etc. This can lead to fraud, loss of privacy/secrecy, phishing, etc. It is limited to some unknown and possibly major extent by HTTPS and HSTS, especially for traffic originating from browsers.

Wormability In scenarios where mobile devices move between networks, the attack may be wormable, in the sense that a mobile device reaching a new network could have a previously spoofed DNS record in its internal DNS cache (received from a vulnerable `dnsmasq` resolver). Accessing this domain is enough to trigger an additional attack on the new network if the attacker wishes (and `dnsmasq` is used).

DDoS and Reverse DDoS Exploiting a large amount of `dnsmasq` resolvers can be used to mount a massive DDoS attack, by leveraging the cache poisoning attack to inject malicious JavaScript when the user surfs to one of the poisoned domains. The original website will be shown, tricking the user in to thinking they are browsing to the authentic site.

In the scenario described thus far, malicious actors substantially increase the amount of traffic forwarded to a website. However, they can also do the reverse - deny service by *reducing* the amount of traffic. In this scenario, malicious actors exploit the vulnerability and redirect users to an incapable server (i.e., server which does not speak HTTP). Those actors can then extort the website owner in return for stopping the attack. We call this type of attack *reverse DDoS*.

3 Cache Poisoning: The Vulnerabilities

The cache poisoning attack is possible due to several different vulnerabilities, all are exploited using a similar setup and have the same outcome of rouge DNS cache entries being cached by `dnsmasq` for an arbitrary amount of time.

The vulnerabilities all reduce the entropy of identifiers TXID (Transaction ID) and source port and their combination, which should consist of 32 non-predictable bits in total, according to [8]. The vulnerabilities need to be combined in order to produce a feasible attack.

3.1 CVE-2020-25684: TXID-Port Decoupling

`dnsmasq` limits the number of unanswered queries that are forwarded to an upstream server. By default, the maximum number of forwarded queries allowed is 150. This value can be configured by the user. Internally, forwarded queries are represented using the `frec` (forward record) structure. Each `frec` is associated with the transaction ID of the forwarded query. This field needs to be correctly guessed by an off-path attacker. `frec`s are removed when an answer has been accepted or certain number of seconds elapsed.

By default, the number of sockets used for forwarding is limited to 64. Each forward socket is bound to a random source port. This means that `dnsmasq` uses source port randomization (SPR) mitigation by default.

In theory, the randomization of TXID and source port should give a 32-bit entropy to each forwarded query. In practice, it is less. This is because `dnsmasq` multiplexes multiple TXIDs on top of one port and does not link each port to a specific TXID. Consequently, the attacker only needs to guess one of 64 ports and the correct TXID (rather than guessing one exact port and one exact TXID). This leads to a practical entropy of only 26 bits ($\frac{2^{32}}{64}$) which is lower than 32 bits, but still not very useful for an attacker. As you

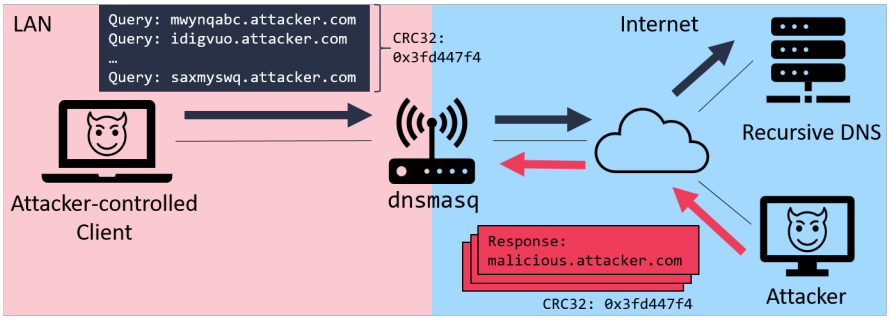


Figure 2: CRC32 poisoning scenario.

will see later in this paper, this vulnerability can be combined with others to produce a feasible and more powerful attack.

3.2 CVE-2020-25685: Weak `frec` Identification

Except for guessing the correct source port, in order to poison the cache, the attacker's fake response needs to match one of the `frec`s currently opened. A response matches an `frec` if the transaction IDs match and the question section match (i.e., the response answers a question that was previously asked).

Instead of storing the entire query's question section in memory, `dnsmasq` records only the *hash* of the question section. This hash value is saved in the corresponding `frec` at the time of query submission.

If `dnsmasq` is not compiled with DNSSEC support, it uses a custom CRC32 algorithm as a hash function. The problem is that CRC32 is *not* a cryptographically secure hash function; therefore this approach is not reliable: an attacker can reply with a specially crafted response, the CRC32 of which is the same as the CRC32 of a query for another name. Making use of this fact, an attacker can generate multiple queries, all with the same CRC32 but a different name being queried. The attacker then responds with an answer whose name has the same CRC32 as all of the requests.

For example, in Figure 2, the attacker-controlled client queries `dnsmasq` for several domains, all of which have the same CRC32 value (according to `dnsmasq` CRC32 implementation). In her response, the attacker places a domain name whose CRC32 value equals to the CRC32 value of the queries.

It is worth mentioning that if `dnsmasq` is compiled with DNSSEC support, it uses SHA-1 for query identification instead of CRC32. A collision attack on SHA-1 has been demonstrated in 2017, and SHA-1 is not considered as a cryptographically secure hash function anymore. To successfully execute the attack with SHA-1, the attacker needs to find 150-300 inputs which hash to the same value. We are not aware of a technique that makes this task feasible currently.

3.3 CVE-2020-25686: Multiple outstanding requests for the same name

Another issue is that `dnsmasq` will incautiously create multiple `frec`s (open requests) with the *same* name being queried. It will forward a request for *each* and will match a correct response to *any* of them. This issue allows to launch a successful attack even if `dnsmasq` uses a cryptographically secure hash function. However, it is likely that some stub resolvers (such as in browsers) will implement a query deduplication behavior that may throttle the rate of queries sent. Therefore, for scenarios where the attacker is not in complete control over the client (e.g., runs a malicious JavaScript in the browser) it is preferable to go with the CRC32 route.

Note that if `dnsmasq` is compiled with DNSSEC, the attack can still be performed if an attacker is able to query the same domain name multiple times, such as in an open instance of `dnsmasq` or control of a machine in the LAN. Table 1 summarizes the conditions.

These last 2 vulnerabilities, in tandem with the decoupling vulnerability, lead to a situation where the attacker needs to get any one of the ports right and *any one of the TXIDs right*. This is because any response which passes the port and TXID checks will pass the question section hash check - thus matches an `frec`.

The weak `frec` identification and decoupling vulnerabilities lead to a practical entropy of $\frac{2^{32}}{150 \cdot 64}$ which is approximately 2^{19}, or approximately half a million queries for cache poisoning. This is a practical amount

Outcome	DNSSEC not compiled	DNSSEC compiled	DNSSEC compiled with validation enabled[3]
Buffer overflow	None	None	CVE-2020-25681 CVE-2020-25682 CVE-2020-25683 CVE-2020-25687
Cache poisoning from browser[1][2]	CVE-2020-25684 CVE-2020-25685	None	None
Cache poisoning from host[1]	CVE-2020-25684 CVE-2020-25685 CVE-2020-25686	CVE-2020-25686	CVE-2020-25686

[1] If caching is disabled (e.g. with the command line option `-c 0`), the cache poisoning vulnerabilities can be used to inject a fake response, appears to be coming from trusted upstream server. This can be used to exploit a buffer overflow or to poison a single `dnsmasq` client.
[2] Some browsers, like Safari. Does not work on Chrome.
[3] E.g. with the `--dnssec` command line option.

Table 1: Attack prerequisites based on DNSSEC compilation and its validation

of packets to send, and takes somewhere between seconds to < 5 minutes over the Internet to exploit using a single attack machine, depending on network conditions and setup. We believe with careful optimization, the attack can be performed in less than 1 minute in most cases[2]. A `dnsmasq` configuration with several upstream servers may raise entropy slightly.

3.4 No Response Verification in DNS Forwarders

An attacker can insert malicious entry into the cache of `dnsmasq` using CNAME records. To illustrate the technique, consider a victim which asks to resolve the domain `www.example.com` (type A), and an attacker which responds with these two answers:

```
www.example.com   CNAME   www.bank.com
www.bank.com      A       6.6.6.6
```

One could expect a DNS server to not believe every answer it got. Indeed, recursive resolvers (such as BIND) employ some form of verification to answers they receive. The set of rules that govern this verification is commonly known as bailiwicks. In the case of the malicious answer shown above, the A record for `www.bank.com` wouldn't be inserted into BIND's cache. Instead, when it constructs the response for the original query, BIND will query for `www.bank.com` (or use a cached entry) to get an authoritative answer for the domain.

In spite of this, as a DNS forwarder, `dnsmasq`, reasonably, believes every response received from its upstream DNS server, and doesn't perform response verification. As response verification obligates contacting authoritative servers, if `dnsmasq` were to implement proper verification, it would effectively have to do the recursive resolution by itself, repeating the work of its upstream server and making query forwarding redundant[3].

The outcome is that when the victim will access her bank's website, `www.bank.com` will be resolved to an attacker-controlled server (`6.6.6.6`), redirecting user traffic to the attacker. This can be exploited together with the decoupling and weak `frec` identification vulnerabilities to poison the cache with up to 9 domain names at once: The maximum length of CNAME chain is 10; The first CNAME must be the same domain name as in the question section[4], the other 9 CNAMEs may contain arbitrary domains. `dnsmasq` will also overwrite *existing* cache entries for the domains being poisoned, regardless of TTL.

[2] The main bottleneck is probably the device's ability to handle responses.
[3] there could be a middle-way of re-asking the upstream server for every name, but this does not make total sense, except in the specific context of the described vulnerabilities.
[4] for more effective attack, this domain will be attacker-controlled, hence "wasting" one CNAME.

We acknowledge that this is not a vulnerability per se, and moreover is reasonable behavior, though it magnifies the attack and similar types of attacks.

4 Cache Poisoning: Attack Details

We confirmed the attack in two scenarios: `dnsmasq` in LAN with attacker-controlled machine (Figure 1b) and `dnsmasq` in LAN from browser (Figure 1c). The scenario of open `dnsmasq` instances was confirmed indirectly as part of the first scenario.

Registering a domain All of these scenarios require that the attacker controls a domain, over which the attack will be based. For this reason, we registered a domain on the Internet and deployed a name server of our own for it. For the purposes of this section, we will refer to the controlled domain as `attacker.com`.

Source IP spoofing All of the attack scenarios require that the attacker has the ability to send IP packets with a spoofed source address. When the attack is performed over the Internet, this requirement suggests finding an ISP which does not restrict sending spoofed packets. Unfortunately, currently these ISPs are easy to find [4]. We demonstrated this by purchasing a plan to use a hosted server in one of these ISPs and sending packets with a spoofed source IP address to another server on a different, more popular, host and ISP.

When the attacker has complete control over a machine on the LAN – the attack can be fully executed locally. In this scenario, this is usually the case that no restriction applies and the attack can be performed with the highest speed.

For convenience reasons, when running our proof-of-concept, which required sending spoofed server responses over the Internet, we used a middle-box which rewrites the source address so that the packets appear to be spoofed.

Generating CRC32 collisions For reasons discussed in § 3, we chose to exploit the CRC32 issue (present if `dnsmasq` is not compiled with DNSSEC). `Dnsmasq` uses custom CRC32 implementation to hash the entire question section of the request/response. For example, a request for `www.bank.com` type `A` class `IN` is hashed using:

$$\mathsf{CRC32}(\texttt{b"www.bank.com\x00\x01\x00\x01"})$$

By the properties of the CRC method, given two inputs x, y such that $\mathsf{CRC32}(x) = \mathsf{CRC32}(y)$, it is possible to extend them with a common suffix s so that $\mathsf{CRC32}(x||s) = \mathsf{CRC32}(y||s)$.

This property suggests that if we can find a set of colliding strings for the first part of the domain name only, then we can append an appropriate suffix later.

We used an SMT solver, specifically `Z3` [12], [5] to find multiple preimages for a specific value – the CRC32 value of the string `"malicious"`. We configured the solver to consider inputs as valid if they contain lowercase ASCII letters or a dot.

This step only needs to be done once, as a pre-processing step. In our case, we calculated over $300,000$ strings, all of them have the same CRC32 value as the string `"malicious"`.

Attack workflow The attack setup consists of a device running `dnsmasq` and two servers: one on the LAN and the other is on the Internet (AWS EC2 free-tier instance).

The server on the LAN generates queries repeatedly for domains with the same CRC32 value. The goal is to reach and maintain the maximum possible outstanding requests (150 by default).

While `dnsmasq` forwards the queries to its upstream server (say, `8.8.8.8`), the server on the Internet sends spoofed responses, with the intention that one of the fake responses would be accepted before the authentic response.

[5] When looking for an easy way to calculate CRC32 collisions, we encountered http://lists.thekelleys.org.uk/pipermail/dnsmasq-discuss/2014q1/008053.html in which, several years ago, Julian Bangert pointed out that CRC32 may have security implications, but not the full flow of an attack. We rewrote the relevant code for our full exploit.

Device	Version (`dnsmasq`)	Vulnerable?	DNSSEC compiled?
Cisco RV160W	2.78	Yes	No
Netgear R7000	2.78	No*	No
OpenWRT 18.06 and 19.07	2.80	Yes	No**
Check Point 1550 V-80	2.78	No*	No

* Caching is disabled with the command line option `-c 0`.
** Unless the `dnsmasq-full` OpenWRT package is installed (not by default).

Table 2: Tested devices

To increase the probability of success, the attacker aspires to increase the time taken for the upstream server to return a response. To achieve this, we arrange for the following:

1. *Always querying for different domains.* We have over $300,000$ domains all with the same CRC32 value. This amount suffices.
2. *Always querying for non-existent domains.* So that they won't be in the cache, and further result in a query sent to the authoritative name server (of which we control).
3. *Delay the response from the authoritative name server as much as possible.* As a simple way to achieve this, we shut down the authoritative server's DNS software. This can likely be greatly improved upon to generate longer response times using different delay methods.

These strategies result in a window of opportunity of 2 seconds, which was pretty good for our purposes. We confirmed that the attack takes between seconds to ≤ 5 minutes to complete in most cases.

Attack from browser Another scenario we verified is executing the attack from a browser. The rules of the game here are that we can trick the victim in to accessing an attacker-controlled website, causing his device to execute malicious JavaScript code.

To make the browser generate many DNS queries, we used AJAX requests (via `XMLHttpRequest`). We tested our malicious JavaScript on Safari in IOS. The result is that Safari generates queries in an adequately high pace. However, it tries to resolve the domain with type `A` and `AAAA`. The implications of this are that instead of 150 useful outstanding requests (`frecs`), we only have 75 (due to the query type being hashed along with the queried domain).

In Firefox it may be possible to use malicious browser extension to generate queries using the JavaScript API `browser.dns.resolve()`. We did not verify this attack vector.

Per our assessment, the Chrome browser limits the amount of outstanding DNS lookups at any given time. This is probably due to having a fixed number of worker threads performing blocking DNS lookups [13]. Consequently, executing a successful attack using the Chrome browser requires more machines in the attacker control.

The behavior of Edge/IE was not determined.

Tested devices `dnsmasq` versions `2.78` to `2.82` (newest at time of disclosure) are all affected by the vulnerabilities. Older versions prior to `2.78` were not tested.

We confirmed the attack on several devices and vendors. Table 2 summarizes the results. Our findings show that some Cisco VPN routers are affected as well as the popular firmware OpenWRT available for hundreds of devices and models. We also tested devices of Netgear and Check Point and interestingly we found that they are using `dnsmasq` with a configuration which disables caching. Therefore, our attack impact these devices only marginally. It is possible that other models of these vendors are configured to use caching, and consequently will be affected. For all tested devices, `dnsmasq` was not compiled with DNSSEC support. Therefore, they are all vulnerable to CRC32 attack flavor.

5 Analysis

Let us briefly sketch a probabilistic analysis of the expected number of packets until success.

Assumptions:

- Spoof packet size (including IP header) is $S = 90$ bytes.
- Time until authentic response arrives $W = 2$ seconds [6].
- Attacker bandwidth (bits/second) is $R = 100$ Mbits/s.
- Number of outstanding queries is $K = 150$.
- Number of source ports $P = 64$.
- Number of valid tuples (port, TXID) is $N = 64512 \times 65536$.[7]

The window of opportunity for an attack is 2 seconds, with which the attacker can send

$$L = W \cdot \frac{R}{8 \cdot S} = 2 \cdot \frac{100\text{M}}{8 \cdot 90} = 291270$$

spoofed packets per second.

The probability that the attacker fails to match a response to a forwarded query is given by:

$$P_{fail} = \prod_{i=0}^{L-1} \left(1 - \frac{K \cdot P}{N - i}\right)$$

When $L \approx \frac{N}{K \cdot P}$, the probability for success is 50%.

In practice we found that we are largely limited by the rate in which packets are received by the network interface. We experienced packet drops by the interface. Consequently, the *effective* bandwidth available for the attacker is limited by the device's ability to process network packets at high rate. This differs from device to device, and so the more processing capability the device has, the faster it can be exploited.

6 DNSSEC Validation Vulnerabilities

The main defence against forgery attacks on the DNS infrastructure is a set of specifications known as DNSSEC [14] [15] [16]. These security extensions provide resolvers with a cryptographically secure way to authenticate data, verify its integrity and prove domain non-existence. DNSSEC establishes a chain of trust from the verified domain up to the root DNS servers. It features key distribution facility by using designated DNS records, and requires the resolver to establish a trust anchor by external means. DNSSEC does not provide transport-layer security; This is provided by DNS-over-TLS [7] or DNS-over-HTTPS [5] specifications.

`dnsmasq` can be configured as a *validating resolver* by ensuring that (a) `dnsmasq` is compiled with DNSSEC support[8], and (b) DNSSEC validation is enabled with the option `--dnssec`. If those two conditions are true, then `dnsmasq` is susceptible to pre-DNSSEC-validation high severity heap-based buffer overflow vulnerabilities found in its latest version at the time of disclosure (`2.82`). The vulnerabilities can be triggered by sending a crafted response packet to an open request (`frec`). This can be combined with the cache poisoning attack to make the exploit easier and potentially mount a remote code execution attack over the device running `dnsmasq`.

Below we include a brief summary of the vulnerabilities and how to trigger them. We did not attempt full exploitation of the vulnerabilities. To verify the vulnerabilities, we used the following command line:

[6] assuming naïve implementation

[7] $65536 - 1024 = 64512$ ports, because ports 0 to 1023 are reserved.

[8] `HAVE_DNSSEC` should be defined during compilation

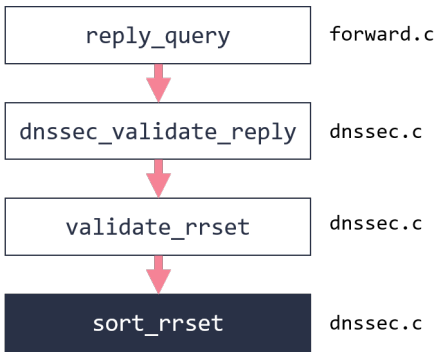


Figure 3: Vulnerable flow

```
sudo ./src/dnsmasq -d --dnssec
↪  --trust-anchor=.,20326,8,2,E06D44B80B8F1D39A95C0B0D7C65D08458E880409BBC683457104237C7F8EC8D --no-resolv
↪  --server=X.Y.Z.W
```

Note: In the following code snippets, the line numbers in each code snippet are the original for version `2.82`.

6.1 `dnsmasq` Internals

Invoking `sort_rrset` The vulnerabilities reside in the `sort_rrset` function in the file `dnssec.c`. Figure 3 shows the flow of functions en route to `sort_rrset`. The function `reply_query` is called whenever a response is accepted (from the upstream server). The functions `dnssec_validate_reply` and `validate_rrset` have less importance to us, but we still need to make sure we pass some checks on our way to `sort_rrset`.

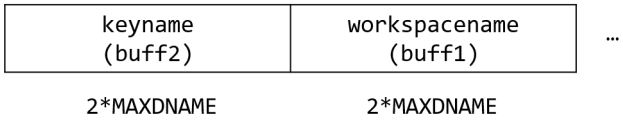
To pass the checks, we can pick some RR type, say T, and include (in the malicious response) an RRSIG record specifying `typecovered=`T and `signername="."`. In addition, at least two resource records of type T should be included (otherwise no sorting is needed!). One of the conditions that `explore_rrset` ensures is that the owner name of the RRSIG and the other RRs is equal, so this condition must be met as well.

`sort_rrset` Internals The `sort_rrset` function is responsible for sorting a given RRset into a canonical order, as required by DNSSEC validation procedure [16]. Its signature is:

```
283 static int sort_rrset(struct dns_header *header, size_t plen, u16 *rr_desc, int rrsetidx,
284                         unsigned char **rrset, char *buff1, char *buff2)
```

We can see it accepts the response packet (`header`) along with the packet length (`plen`). `rrset` is a pointer to an array of RRs in the set, and `rrsetidx` is the number of RRs in the set. `rr_desc` is a pointer to a *descriptor* of the RR type associated with the RRset. The descriptor specifies how to interpret each part of the `RDATA` field: whether it's a domain name or raw data. More about this descriptor later.

Finally, there are two buffers, `buff1` and `buff2`, which serve as workspace buffers for the sorting routine. Both buffers are allocated relatively in the beginning of the program, and they are `daemon->workspacename` and `daemon->keyname`, respectively. Those will be the buffers we'll overflow, so let's examine how they are laid out in memory:



`MAXDNAME` is a constant value equals 1025, so `2*MAXDNAME` equals 2050.

As already pointed out, `sort_rrset` is responsible for ordering the RRs in the given `rrset` in canonical order. The underlying sorting algorithm implemented in `dnsmasq` is bubble sort. The function itself is quite

convoluted, so we'll present the high level structure first, and provide snippets of code as the explanation progresses. The high level structure is shown in the following snippet:

```
int swap, quit, i, j;
do {
    for (swap = 0, i = 0; i < rrsetidx-1; i++) {
        /* read rrset[i] */
        /* set p1 to the beginning of the RDATA field of rrset[i] */
        end1 = p1 + rdlen1;

        /* read rrset[i+1] */
        /* set p2 to the RDATA of rrset[i+1] */
        end2 = p2 + rdlen2;

        dp1 = dp2 = rr_desc;

        for (quit = 0, left1 = 0, left2 = 0, len1 = 0, len2 = 0; !quit;) {
            /* determine if swap is needed */
            /* if so, set swap=1. */
        }
    }
} while(swap);
```

The high-level structure contains 3 nested loops. The outer-most loop is executed whenever the inner-most loop sets `swap` to 1. In each iteration of the outer-most loop, each pair of records in the RRset is considered.

We will focus on the inner-most `for` loop, where it is determined if a swap is needed. To understand the inner `for` loop, we need to take a deeper look at the rules that govern the sorting. The rules for canonicalizing resource records and sorting RRsets are described in [15]. A subset of these is provided here (emphasis added):

> "For the purposes of DNS security, RRs with the same owner name, class, and type are sorted by treating the *RDATA portion* of the canonical form of each RR as a *left-justified unsigned octet sequence in which the absence of an octet sorts before a zero octet.*"

From the specification we learn that (a) an RRset is a collection of resource records with the same owner name, class and type; (b) the field governs the sorting is the `RDATA` field; and (c) the resource record must be in canonical form. This canonical form include: all names are expanded (no compression) and converted to use lowercase letters. (Note that the structure of the `RDATA` field depends on the resource record type. For example, records of type `A` carry IPv4 address, while `MX` records carry preference value and a mail server name.)

In `dnsmasq`, there is no separate function which canonicalizes a resource record. Instead, the data is converted to canonical form as it is read from the `RDATA` portion of the packet. The function responsible for this task is `get_rdata`:

```
228 static int get_rdata(struct dns_header *header, size_t plen, unsigned char *end, char *buff, int bufflen,
229         unsigned char **p, u16 **desc)
```

This function reads data from `*p`, converts it to canonical form, and stores the result in `buff`. To know how to interpret the data, i.e. name or raw data, the descriptor `desc` is used.[9] `get_rdata` is also responsible for advancing `*p` and `*desc`. This is why they are passed as double pointers. The function returns the number of bytes copied into the buffer. If the return value is 0, the remaining packet data should be used as raw data (any data which might have been copied into `buff` should not be considered). This subtle behavior leads to the first vulnerability - CVE-2020-25681.

6.2 CVE-2020-25681: Heap-based buffer overflow with arbitrary overwrite

Recall that triggering `sort_rrset` requires picking a record type T. For this vulnerability to work, T should be picked from the following set: NS, MD, MF, SOA, MB, MG, MR, PTR, MINFO, MX, RP, AFSDB,

[9] The use of descriptors enables handling different record types in a uniform way.

RT, PX, NXT, KX, SRV, DNAME. What is special about these types is that their `RDATA` field includes a name. If this condition is met, it is possible to trigger a heap-based buffer overflow with arbitrary overwrite. This can be done by causing `get_rdata` to return 0 on the first/second RR, thus affecting `buff1` or `buff2` respectively.

To understand why this condition results in memory corruption, consider the following code snippet taken from the inner `for` loop of `sort_rrset`:

```
315 if ((len1 = get_rdata(header, plen, end1, buff1 + left1, (MAXDNAME * 2) - left1, &p1, &dp1)) == 0)
316   {
317     quit = 1;
318     len1 = end1 - p1;
319     memcpy(buff1 + left1, p1, len1);
320   }
321 len1 += left1;
```

We see that if `get_rdata` returns 0, the remaining data from the resource record's `RDATA` is copied to `buff1`. The issue is that the end pointer (`end1`) is computed based on the RR's `RDLENGTH` field, which is attacker controlled. The size of `buff1` is fixed: `MAXDNAME*2` (2050). It can be overflowed by including an RR with long `RDATA` field (longer than 2050). Such records could be sent over UDP as the use of DNSSEC requires the use of EDNS(0) extensions, which among other things, increase the maximum packet size allowed. The largest EDNS(0) UDP packet which is supported by `dnsmasq` is 4096 by default. This size is sufficient to cleanly overwrite heap memory.

The return value of `get_rdata` can be coerced to be 0 when the descriptor (`desc`) represents a name (`d == 0`). This is due to the following logic in `get_rdata`:

```
250    if (d == 0 && extract_name(header, plen, p, buff, 1, 0))
251      /* domain-name, canonicalise */
252      return to_wire(buff);
253    else
254      {
255        /* plain data preceding a domain-name, don't run off the end of the data */
256        if ((end - *p) < d)
257          d = end - *p;
258
259        if (d != 0)
260          {
261            memcpy(buff, *p, d);
262            *p += d;
263          }
264
265        return d;
266      }
267  }
```

We see that if `d == 0` but `extract_name` return 0, the function returns 0. Making `extract_name` return 0 can be done in a few ways - including a long name that will exceed `MAXDNAME` or crafting a malicious compression pointer loop.

6.2.1 Artifact: Infinite loop

Recall that the role of `get_rdata` is to extract data from the resource record in canonical form. The extracted data should be compared to determine the canonical order between the two RRs considered.

The following logic from `sort_rrset` is responsible for comparison:

```
334   if (len1 > len2)
335 left1 = len1 - len2, left2 = 0, len = len2;
336   else
337 left2 = len2 - len1, left1 = 0, len = len1;
338
339   rc = (len == 0) ? 0 : memcmp(buff1, buff2, len);
```

`len1` and `len2` are the lengths of `buff1` and `buff2`, respectively. The first `len=min(len1,len2)` bytes of the two buffers are compared using `memcmp`.

Based on the `rc` variable, the next iteration is determined. If `rc` is greater than 0, a swap occurs, the inner loop finishes and the next pair of RRs is considered. If `rc` is less than 0, there is no swap and the inner loop finishes. The case where `rc` equals 0 is slightly more involved and is discussed below.

If the heap overflow vulnerability is triggered on `buff2`, it is possible to affect the state of the other buffer (`buff1`). (Recall how they're laid out in memory.) This affects the comparison result, and could lead to infinite loop. Here's how:

Consider two records in the RRset. In the first iteration of the inner loop, there will be overflow from `buff1` into the heap. In addition, the data in the first record will cause `memcmp` to return `rc` > 0. This situation leads to a swap, so the outer loop is performed a second time. This time, `p1` and `p2` exchanged roles, so the overflow will be from `buff2` into `buff1`. When the data with which the overwrite is performed is carefully chosen, it is possible to cause `memcmp` to return `rc` > 0 again, thereby swapping once more, returning to the initial state. The process will continue forever.

6.3 CVE-2020-25682: Heap-based buffer overflow with null bytes

The vulnerability discussed on the previous section was relatively simple. The infinite loop artifact demonstrated the intricate connections between the different functions and variables. A second vulnerability, discussed now, exploits the interconnection between `sort_rrset` and `get_rdata` from a different angle.

As already established, when the descriptor says that the current part of the RR's `RDATA` is a name, the function `extract_name` is called. `extract_name` extracts a name from the packet and converts it from wire-format to "presentation"-format. Presentation format is similar to the wire format, only that the length bytes are replaced by periods, all letters are converted to lowercase and certain designated characters are escaped. If the name is extracted (and converted) successfully, a second conversion is used to transform the result from presentation-format to a *canonical* wire-format. The second conversion is performed by `to_wire`.

The escaping performed by `extract_name` occurs only when `dnsmasq` is compiled with DNSSEC support and turn on DNSSEC validation. You can see it in the following logic from `extract_name` (in `rfc1035.c`):

```
80        else if (label_type == 0x00)
81          { /* label_type = 0 -> label. */
82            namelen += l + 1; /* include period */
83            if (namelen >= MAXDNAME)
84              return 0;
85            if (!CHECK_LEN(header, p, plen, l))
86              return 0;
87
88            for(j=0; j<l; j++, p++)
89              if (isExtract)
90                {
91                  unsigned char c = *p;
92  #ifdef HAVE_DNSSEC
93                  if (option_bool(OPT_DNSSEC_VALID))
94                    {
95                      if (c == 0 || c == '.' || c == NAME_ESCAPE)
96                        {
97                          *cp++ = NAME_ESCAPE;
98                          *cp++ = c+1;
99                        }
100                     else
101                       *cp++ = c;
102                   }
103                 else
104 #endif
```

`label_type` of 0 means a regular label (as opposed to compressed labels). We see that if the accumulated name length (tracked by `namelen`) exceeds `MAXDNAME`, the function fails. (Keep in mind that the name buffer size is `MAXDNAME*2`.) After another sanity check, the function iterates over every character in the label and copies it to the destination buffer (`cp`). However, if the character is one of `\0` (null byte) or `'.'` (period)

or `NAME_ESCAPE` (which expands to `0x01`), then *two* characters are written into the destination buffer (lines 97-98). This is the reason behind setting the name buffer size to twice `MAXDNAME`.

The code is perfectly fine in cases where the destination buffer size is at least `MAXDNAME*2`. Otherwise, an out-of-bounds overwrite could be caused due to improper length validation during escaping. It turns out that there is one place where the size of destination buffer passed to `extract_name` is not necessarily `MAXDNAME*2`. This situation can take place in `sort_rrset`:

```
312       if ((len1 = get_rdata(header, plen, end1, buff1 + left1, (MAXDNAME * 2) - left1, &p1, &dp1)) == 0)
313     {
314       quit = 1;
315       len1 = end1 - p1;
316       memcpy(buff1 + left1, p1, len1);
317     }
318       len1 += left1;
```

We already saw this code before. Notice how the destination buffer passed to `get_rdata` (and subsequently to `extract_name`) is `buff1 + left1`. In other words, an offsetted buffer is passed. The assumption that `extract_name` makes about the size of its destination buffer no longer holds true when the offset (`left1`) is greater than 0.[10]

Remember how the extracted name is then converted to a canonical wire-format by `to_wire`. When `to_wire` reaches an escape character, it will shift the entire buffer one byte to the left and undo the escaping operation. While doing this it will keep copying the trailing null-byte:

```
43    static int to_wire(char *name)
44    {
...       // ....
55        for (q = p; *q; q++)
56          *q = *(q+1);        /* shift left (copy trailing null byte) */
57        (*p)--;               /* undo escaping */
```

The end result is that the arbitrary bytes written out-of-bounds by `extract_name` are replaced with null bytes after `to_wire` is executed. Consequently, this primitive gives a heap-based buffer overflow with null bytes.

It remains to be shown that it is possible to arrange for a large offset (`left1`/`left2`) that will cause memory corruption. This is a little tricky to get right. To explain the set of conditions which needs to be met to reach this situation, consider the inner `for` loop of `sort_rrset`:

```
310     for (quit = 0, left1 = 0, left2 = 0, len1 = 0, len2 = 0; !quit;)
311       {
312         if (left1 != 0)
313       memmove(buff1, buff1 + len1 - left1, left1);
314
315         if ((len1 = get_rdata(header, plen, end1, buff1 + left1, (MAXDNAME * 2) - left1, &p1, &dp1)) == 0)
316       {
317         quit = 1;
318         len1 = end1 - p1;
319         memcpy(buff1 + left1, p1, len1);
320       }
321         len1 += left1;
322
323         if (left2 != 0)
324       memmove(buff2, buff2 + len2 - left2, left2);
325
326         if ((len2 = get_rdata(header, plen, end2, buff2 + left2, (MAXDNAME *2) - left2, &p2, &dp2)) == 0)
327       {
328         quit = 1;
329         len2 = end2 - p2;
330         memcpy(buff2 + left2, p2, len2);
```

[10] Due to memory allocator size alignment, the offset might actually need to be larger to overwrite the next heap chunk. Our experiments show that an offset of at least 18 is sufficient.

Preference	MAP822 (name)	MAPX400 (name)
2 bytes	variable	variable

Figure 4: PX record RDATA field

```
331     }
332         len2 += left2;
333
334         if (len1 > len2)
335     left1 = len1 - len2, left2 = 0, len = len2;
336         else
337     left2 = len2 - len1, left1 = 0, len = len1;
338
339         rc = (len == 0) ? 0 : memcmp(buff1, buff2, len);
340
341         if (rc > 0 || (rc == 0 && quit && len1 > len2))
342     {
343       unsigned char *tmp = rrset[i+1];
344       rrset[i+1] = rrset[i];
345       rrset[i] = tmp;
346       swap = quit = 1;
347     }
348         else if (rc == 0 && quit && len1 == len2)
349     {
350       /* Two RRs are equal, remove one copy. RFC 4034, para 6.3 */
351       for (j = i+1; j < rrsetidx-1; j++)
352         rrset[j] = rrset[j+1];
353       rrsetidx--;
354       i--;
355     }
356         else if (rc < 0)
357     quit = 1;
358       }
359   }
```

We see that the offsets, `left1` and `left2`, are set on lines 334-337 if there is a discrepancy between `len1` and `len2`. On line 339 a comparison between `buff1` and `buff2` is executed. Based on the value of `rc` an action is taken (lines 341-358): swap, remove duplicate, quit loop or do nothing (proceed with the next iteration). Therefore, to reach a situation where `left1` or `left2` are positive during the second iteration, the value of `quit` must be zero *and* `rc` must be zero.

The tricky part is that the comparison on line 339 compares between buffers in wire-format. It is easy enough to cause a discrepancy between `len1` and `len2` by specifying a name which is longer than the other. However, the comparison takes the label length bytes into account, which are different if the names are different. The naïve attempt fails.

The key insight is that bytes not participating in the buffer comparison are `memmove`'d in the next iteration. It is possible to exploit this behavior when records of type PX [1] are used.

PX records are now obsolete, but still accepted by `dnsmasq`. Their original purpose is irrelevant for this discussion, only the structure of their `RDATA` field, which is shown in Figure 4. The `MAP822` and `MAPX400` fields are domain names. The descriptor for this record type specifies 2 raw bytes (for the preference value) followed by two domains.

It is possible to control the length returned by `get_rdata` when leading raw bytes are processed. This is done by specifying an `RDLENGTH` value which is small. An example illustrates this better.

Consider the two records shown in Figure 5. On the first iteration of the inner loop, two bytes are read for the first record and only *one* byte is read for the second record. This is due to specifying `RDLENGTH` value of 1 for the second record. Here's the logic from `get_rdata` which explains this behavior:

```
250     if (d == 0 && extract_name(header, plen, p, buff, 1, 0))
251       /* domain-name, canonicalise */
```

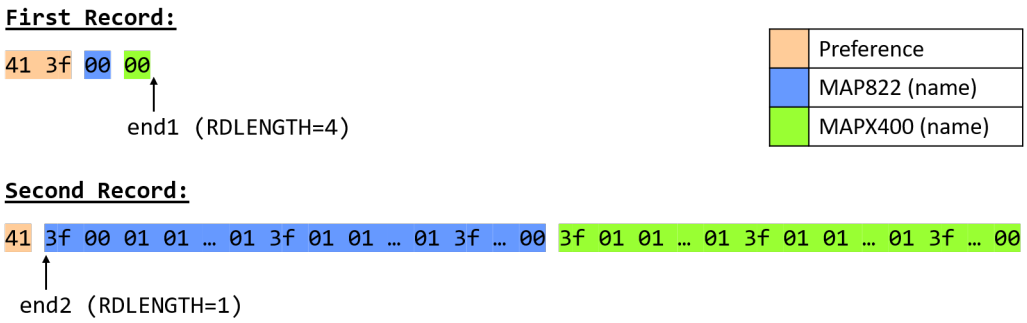


Figure 5: Two PX records. The second is malformed since the `RDLENGTH` value is 1.

```
252       return to_wire(buff);
253     else // NOTE: this branch is taken since d==2
254       {
255         /* plain data preceding a domain-name, don't run off the end of the data */
256         if ((end - *p) < d)
257           d = end - *p;
258
259         if (d != 0)
260           {
261             memcpy(buff, *p, d);
262             *p += d;
263           }
264
265         return d;
266       }
267   }
```

Initally, `d` equals 2 and the `else` branch is taken. Since we specified `RDLENGTH` of 1, `d` will be updated on line 257 to be 1 (instead of 2). This value is returned from `get_rdata`.

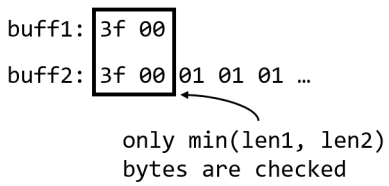
Back to the inner loop, the `memcmp` call is executed, but only a single byte is checked. In our example, the first byte of the first and second record is `0x41`, so `rc` comes out 0. `left1` is set to $2-1=1$. Since `quit` is not set to 1 in any stage, a second iteration follows.

In the second iteration of the inner loop, the second byte of the first record (`0x3f`) will moved to offset 0 of `buff1`, by the line:

```
312               if (left1 != 0)
313                 memmove(buff1, buff1 + len1 - left1, left1);
```

The first call to `get_rdata` will read a null byte to offset 1 of `buff1` (as specified in the first record in Figure 5). The second call to `get_rdata` will read a name starting from `0x3f` (second record of Figure 5). The first character of this name is null-byte (to match the null-byte read from the first record). Null-bytes embedding is a feature, though it is not crucial for this vulnerability. Note that the name being read from the second record is read after the end pointer. This is possible because `extract_name` performs a bound check on the whole packet buffer rather than on resource records boundaries.

Our records were crafted so that `len2 == 0x401` and `len1 == 1+1 == 2`. This results in two bytes being compared in the `memcmp` call:



Consequently, a third iteration of the inner loop will be taken. Now with a large `left2` value of `0x3ff`.

When the third iteration fires, a second name will be read into `buff2 + left2` (corresponding to `MAPX400` field of the PX RR). This name will contain many characters that need to be escaped, and thus overflowing our buffer.

In this example `buff2` is overflowed. Our findings show that it is also possible to craft a packet such that `buff1` will be overflowed.

6.4 CVE-2020-25683/7: Heap-based buffer overflow with large memcpy

The third and fourth vulnerabilities allow for the execution of a very large `memcpy` (`memcpy` that is called with a negative length; sometimes called "wild copy"). There are two ways to achieve this, both can be said to stem from the same root cause, and are tracked as separate CVEs.

Say we specify an `RDLENGTH` value of 1 as explained previously. `extract_name` only check for packet buffer boundaries (it does not check resource records boundaries), so it will advance the given name pointer (`*p`) past the RR end pointer. This leads to a situation where the difference `end-*p` is negative.

There are two routes to exploit this situation. The first, in `sort_rrset` (CVE-2020-25687):

```
326 if ((len2 = get_rdata(header, plen, end2, buff2 + left2, (MAXDNAME *2) - left2, &p2, &dp2)) == 0)
327   {
328     quit = 1;
329     len2 = end2 - p2; // NOTE: can be negative
330     memcpy(buff2 + left2, p2, len2);
331   }
```

If `get_rdata` returns 0 (section 6.2 discusses how it can be done), then `len2` might end up negative, leading to a wild copy.

The second way is found in `get_rdata` (CVE-2020-25683):

```
250   if (d == 0 && extract_name(header, plen, p, buff, 1, 0)) // NOTE: extract_name returns 0
251     /* domain-name, canonicalise */
252     return to_wire(buff);
253   else
254     {
255     /* plain data preceding a domain-name, don't run off the end of the data */
256      if ((end - *p) < d) // NOTE: signed comparison
257         d = end - *p; // NOTE: will enter here, d will be negative
258
259      if (d != 0)
260      {
261       memcpy(buff, *p, d);  // NOTE: will reach here
262       *p += d;
263      }
264
265      return d;
266     }
```

To reach this situation, the same method of using PX records is used. This time, the `MAPX400` field will contain an invalid name which causes `extract_name` to return 0. On the `else` branch, the value `(end-*p)` is negative. The variable `d` is declared as a signed integer; therefore, the comparison on line 256 is a signed comparison. Since `d` equals 0 and `(end-*p)` is negative, `d` is set to a negative value on line 257. The wild copy happens on line 261.

7 Mitigations

Find out if your device runs `dnsmasq` by consulting the device's vendor or by issuing a chaos query using:

```
dig +short chaos txt version.bind
```

Update your `dnsmasq` software to the latest version (2.83 or above). This is the best and only complete mitigation.

To prevent an attack from the LAN, implement layer 2 security features, such as DHCP snooping and IP source guard.

Workarounds (partial):

- Configure `dnsmasq` not to listen on WAN interfaces if unnecessary in your environment.
- Reduce the maximum queries allowed to be forwarded with the option `--dns-forward-max=<queries>`. The default is 150, but it could be lowered.
- Temporarily disable DNSSEC validation option until you get a patch.
- Use protocols that provide transport security for DNS (such as DoT or DoH). This will mitigate Dnspooq but may have other security and privacy implications. Consider your own setup, security goals, and risks before doing this.
- Reducing the maximum size of EDNS messages will likely mitigate some of the vulnerabilities. This, however, has not been tested and is against the recommendation of the relevant RFC5625.

8 Conclusion

DNS is an Internet-critical protocol whose security greatly affect the security of Internet users. In this paper, we presented 7 vulnerabilities affecting the popular DNS forwarder `dnsmasq`. These issues put networking and other devices at a risk of compromise and affect millions of Internet users which can suffer from the cache poisoning attack and RCE presented. This highlights the importance of DNS security in general and the security of DNS forwarders in particular. It also highlights the need to expedite the deployment of DNS security measures such as DNSSEC, DNS transport security and DNS cookies.

> *It's not DNS...*
> *There's no way it was DNS...*
> *It was DNS.*
>
> - SSBroski

9 References

[1] C. Allocchio, A. B. Bonito, B. A. Cole, S. Giordano, and R. Hagens. Using the Internet DNS to Distribute RFC1327 Mail Address Mapping Tables. RFC 1664, Aug. 1994. URL https://rfc-editor.org/rfc/rfc1664.txt.

[2] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee. Increased dns forgery resistance through 0x20bit encoding. pages 211–222, 01 2008. doi: 10.1145/1455770.1455798.

[3] K. S. Fermin J. Serna, Matt Linton. Behind the Masq: Yet more DNS, and DHCP, vulnerabilities. URL https://security.googleblog.com/2017/10/behind-masq-yet-more-dns-and-dhcp.html.

[4] C. for Applied Internet Data Analysis. State of IP Spoofing. URL https://spoofer.caida.org/summary.php.

[5] P. E. Hoffman and P. McManus. DNS Queries over HTTPS (DoH). RFC 8484, Oct. 2018. URL https://rfc-editor.org/rfc/rfc8484.txt.

[6] P. E. Hoffman, A. Sullivan, and K. Fujiwara. DNS Terminology. RFC 8499, Jan. 2019. URL https://rfc-editor.org/rfc/rfc8499.txt.

[7] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. E. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, May 2016. URL https://rfc-editor.org/rfc/rfc7858.txt.

[8] B. Hubert and R. Mook. Measures for Making DNS More Resilient against Forged Answers. RFC 5452, Jan. 2009. URL https://rfc-editor.org/rfc/rfc5452.txt.

[9] D. Kaminsky. Black ops 2008 – its the end of the cache as we know it. 2008.

[10] S. Kelley. Dnsmasq. URL http://www.thekelleys.org.uk/dnsmasq/doc.html.

[11] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan. Dns cache poisoning attack reloaded: Revolutions with side channels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1337–1350, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417280. URL https://doi.org/10.1145/3372297.3417280.

[12] Microsoft. Z3 Theorem Prover. URL https://github.com/Z3Prover/z3.

[13] T. C. Project. DNS Prefetching. URL https://www.chromium.org/developers/design-documents/dns-prefetching.

[14] S. Rose, M. Larson, D. Massey, R. Austein, and R. Arends. DNS Security Introduction and Requirements. RFC 4033, Mar. 2005. URL https://rfc-editor.org/rfc/rfc4033.txt.

[15] S. Rose, M. Larson, D. Massey, R. Austein, and R. Arends. Resource Records for the DNS Security Extensions. RFC 4034, Mar. 2005. URL https://rfc-editor.org/rfc/rfc4034.txt.

[16] S. Rose, M. Larson, D. Massey, R. Austein, and R. Arends. Protocol Modifications for the DNS Security Extensions. RFC 4035, Mar. 2005. URL https://rfc-editor.org/rfc/rfc4035.txt.

[17] X. Zheng, C. Lu, J. Peng, Q. Yang, D. Zhou, B. Liu, K. Man, S. Hao, H. Duan, and Z. Qian. Poison over troubled forwarders: A cache poisoning attack targeting DNS forwarding devices. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 577–593. USENIX Association, Aug. 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/zheng.

WHO WE ARE

JSOF is a multidisciplinary team focused on solving product cyber security challenges. We are research oriented and focus exclusively on product security. We excel in projects that are complex, time-sensitive, or mission-critical.

CONTACT US

www.jsof-tech.com
info@jsof-tech.com

JSOF