Independent Submission
Internet-Draft
Intended status: Informational
Expires: August 14, 2022

S. Dashevskyi
D. dos Santos
J. Wetzels
A. Amri
Forescout Technologies
February 14, 2022

Common implementation anti-patterns related to Domain Name System (DNS) resource record (RR) processing
draft-dashevskyi-dnsrr-antipatterns-01

Abstract

This memo describes common vulnerabilities related to Domain Name System (DNS) response record (RR) processing as seen in several DNS client implementations. These vulnerabilities may lead to successful Denial-of-Service and Remote Code Execution attacks against the affected software. Where applicable, violations of RFC 1035 are mentioned.

Table of Contents

1. Introduction

Recently, there have been major vulnerabilities on DNS implementations that raised attention to this protocol as an important attack vector, such as CVE-2020-1350, known as "SIGRed", CVE-2020-2705, known as "SAD DNS", and "DNSpooq", a set of 7 critical issues affecting the DNS forwarder "dnsmasq" (<https://thekelleys.org.uk/dnsmasq/doc.html>).

The authors of this memo have analyzed the DNS client implementations of several major TCP/IP protocol stacks and found a set of vulnerabilities that share common implementation flaws (anti-patterns). These flaws are related to processing DNS RRs (discussed in [RFC1035]) and may lead to critical security vulnerabilities.

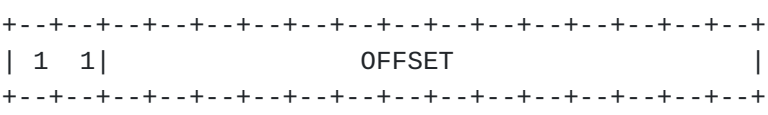
While implementation flaws may differ from one software project to another, these anti-patterns are highly likely to span across multiple implementations. In fact, one of the first CVEs related to one of the anti-patters (CVE-2000-0333) dates back to the year 2000. The affected software is not limited to DNS client implementations, and any software that attempts to process DNS RRs may be affected, such as firewalls, intrusion detection systems, or general purpose DNS packet dissectors (i.e., Wireshark).

[COMP-DRAFT] and [RFC5625] briefly mention some of these anti-patterns, but the main purpose of this memo is to provide technical details behind these anti-patterns, so that the common mistakes can be eradicated.

2. Compression Pointer and Offset Validation

[RFC1035] defines the DNS message compression scheme that can be used to reduce the size of messages. When it is used, an entire domain name or several name labels are replaced with a (compression) pointer to a prior occurrence of the same name.

The compression pointer is a combination of two octets: the two most significant bits are set to 1, and the remaining 14 bits are the OFFSET field. This field specifies the offset from the beginning of the DNS header, at which another domain name or label is located:



The message compression scheme explicitly allows a domain name to be represented as: (1) a sequence of unpacked labels ending with a zero octet; (2) a pointer; (3) a sequence of labels ending with a pointer.

However, [RFC1035] does not explicitly state that blindly following compression pointers of any kind can be harmful [COMP-DRAFT], as the authors could not have had any assumptions about various implementations that would follow.

Yet, any DNS packet parser that attempts to uncompress domain names without validating the value of OFFSET is likely susceptible to memory corruption bugs and buffer overruns. These bugs allow for easy Denial-of-Service attacks, and may result in successful Remote Code Execution attacks.

Pseudocode that illustrates typical domain name parsing implementations is shown below (Snippet 1):

```
 1:uncompress_domain_name(*name, *dns_payload) {
 2:
 3:  name_buffer[255];
 4:  copy_offset = 0;
 5:
 6:  label_len_octet = name;
 7:  dest_octet = name_buffer;
 8:
 9:  while (*label_len_octet != 0x00) {
10:
11:     if (is_compression_pointer(*label_len_octet)) {
12:         ptr_offset = get_offset(label_len_octet,
                                    label_len_octet+1);
13:         label_len_octet = dns_payload + ptr_offset + 1;
14:     }
15:
16:     else {
```

```
17:         length = *label_len_octet;
18:         copy(dest_octet + copy_offset,
                        label_len_octet+1, *length);
19:
20:         copy_offset += length;
21:         label_len_octet += length + 1;
22:     }
23:
24:  }
25:}
```

Snippet 1 - A typical implementation of a function that is used for uncompressing DNS domain names (pseudocode)

Such implementations typically have a dedicated function for uncompressing domain names. Among other parameters, these functions may accept a pointer to the beginning of the first name label within a RR ("name") and a pointer to the beginning of the DNS payload to be used as a starting point for the compression pointer ("dns_payload"). The destination buffer for the domain name ("name_buffer") is typically limited to 255 bytes as per [RFC1035] and can be allocated either in the stack or in the heap memory region.

The code of the function at Snippet 1 reads the domain name label-by-label from a RR until it reaches the NULL octet (0x00) that signifies the end of a domain name. If the current label length octet ("label_len_octet") is a compression pointer, the code extracts the value of the compression offset and uses it to "jump" to another label length octet. If the current label length octet is not a compression pointer, the label bytes will be copied into the name buffer, and the number of bytes copied will correspond to the value of the current label length octet. After the copy operation, the code will move on to the next label length octet.

There are multiple issues with this implementation. In this Section we discuss the issues related to the handling of compression pointers.

The first issue is due to unchecked compression offset values. The second issue is due to the absence of checks that ensure that a pointer will eventually arrive at an uncompressed domain label. We describe these issues in more detail below.

2.1 Compression Pointer Pointing Out of Bounds

[RFC1035] states that "... [compression pointer is] a pointer to a prior occurrence of the same name". Also, according to [RFC1035], the maximum size of DNS packets that can be sent over the UDP protocol is limited to 512 octets.

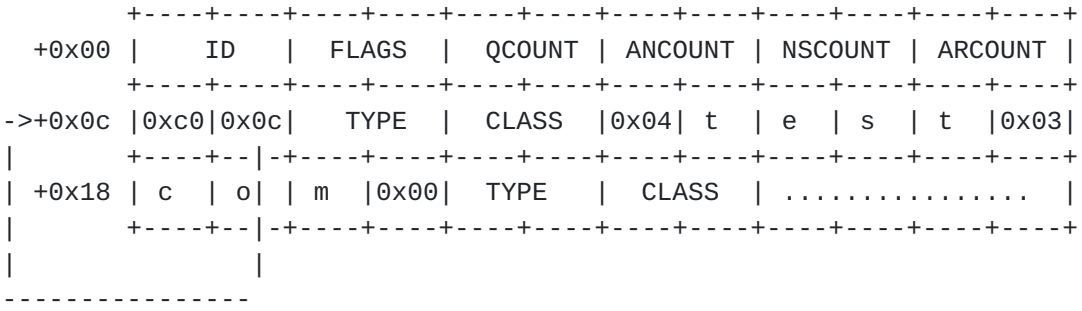
The pseudocode at Snippet 1 violates these constraints, as it will accept a compression pointer that forces the code to read out of the

bounds of a DNS packet. For instance, the compression pointer of 0xffff will produce the offset of 16383 octets, which is most definitely pointing to a label length octet somewhere past the original DNS packet. Supplying such offset values will most likely cause memory corruption issues and may lead to Denial-of-Service conditions (e.g., a NULL pointer dereference after "label_len_octet" is set to an invalid address in memory).

2.2 Compression Pointer Loops

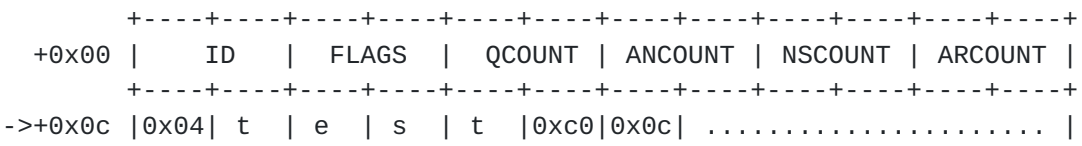
The pseudocode at Snippet 1 allows for jumping from a compression pointer to another compression pointer and it does not restricts the number of such jumps. That is, if a label length octet which is currently being parsed is a compression pointer, the code will perform a jump to another label, and if that other label is a compression pointer as well, the code will perform another jump, and so forth until it reaches an uncompressed label. This may lead to unforeseen side-effects that result in security issues.

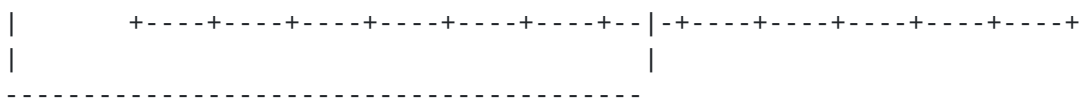
Consider the excerpt from a DNS packet illustrated below:



The packet begins with a DNS header at the offset +0x00, and its DNS payload contains several RRs. The first RR begins at the offset of 12 octets (+0xc0) and its first label length octet is set to the value "0xc0", which indicates that it is a compression pointer. The compression pointer offset is computed from the two octets "0xc00c" and it is equal to 12. Since the implementation at Snippet 1 follows this offset value blindly, the pointer will jump back to the first octet of the first RR (+0xc0) over and over again. The code at Snippet 1 will enter an infinite loop state, since it will never leave the "TRUE" branch of the "while" loop.

Apart from achieving infinite loops, the implementation flaws at Snippet 1 make it possible to achieve various pointer loops that have different effects. For instance, consider the DNS packet excerpt shown below:





With such a domain name, the implementation at Snippet 1 will first copy the domain label at the offset 0xc0 ("test"), then it will fetch the next label length octet, which is a compression pointer (0xc0). The compression pointer offset is computed from the two octets "0xc00c" and is equal to 12 octets. The code will jump back at the offset 0xc0 where the first label "test" is located. The code will again copy the "test" label, and jump back to it, following the compression pointer, over and over again.

Snippet 1 does not contain any logic that restricts multiple jumps from the same compression pointer and does not ensure that no more than 255 octets are copied into the name buffer ("name_buffer"). In fact, the code will continue to write the label "test" into it, overwriting the name buffer and the stack of the heap metadata. In fact, attackers would have a significant degree of freedom in constructing shell-code, since they can create arbitrary copy chains with various combinations of labels and compression pointers.

Therefore, blindly following compression pointers may not only lead to Denial-of-Service as pointed by [COMP-DRAFT], but also to successful Remote Code Execution attacks, as there may be other implementation issues present within the corresponding code.

2.3 Invalid compression pointer check

Some implementations may not follow [RFC1035], which states: "the first two bits [of a compression pointer octet] are ones; this allows a pointer to be distinguished from a label, the label must begin with two zero bits because labels are restricted to 63 octets or less (the 10 and 01 combinations are reserved for future use)". Snippets 2 and 3 show pseudocode that implements two functions that check whether a given octet is a compression pointer: correct and incorrect implementations respectively.

```
1: unsigned char is_compression_pointer(*octet) {
2:     if ((*octet & 0xc0) == 0xc0)
3:         return true;
4:     } else {
5:         return false;
6:     }
7: }
```

Snippet 2 - Correct compression pointer check

```
1: unsigned char is_compression_pointer(*octet) {
2:     if (*octet & 0xc0) {
3:         return true;
4:     } else {
```

```
5:         return false;
6:     }
7: }
```

Snippet 3 - Incorrect compression pointer check

The correct implementation (Snippet 2) ensures that the two most significant bits of an octet are both set, while the incorrect implementation (Snippet 3) would consider an octet with only one of the two bits set as a compression pointer. This is likely an implementation mistake rather than an intended violation of [RFC1035], because there are no benefits in supporting such compression pointer values.

While incorrect implementations alone do not lead to vulnerabilities, they may have unforeseen side-effects when combined with other vulnerabilities. For instance, the first octet of the value "0x4130" represents an invalid label length (65) which is larger than 63 (as per [RFC1035]) and a packet that has this value should be discarded. However, the function shown on Snippet 3 will consider "0x41" to be a valid compression pointer, and the packet may pass the validation steps.

This might give an additional leverage for attackers in constructing payloads and circumventing the existing DNS packet validation mechanisms.

3. Label and Name Length Validation

[RFC1035] restricts the length of name labels to 63 octets, and lengths of domain names to 255 octets. Some implementations do not explicitly enforce these restrictions.

Consider the pseudocode function "copy_domain_name()" shown on Snippet 4 below. The function is a variant of the "uncompress_domain_name" function (Snippet 1), with the difference that it does not support compressed labels, and copies only uncompressed labels into the name buffer.

```
 1:copy_domain_name(*name, *dns_payload) {
 2:
 3:  name_buffer[255];
 4:  copy_offset = 0;
 5:
 6:  label_len_octet = name;
 7:  dest_octet = name_buffer;
 8:
 9:  while (*label_len_octet != 0x00) {
10:
11:     if (is_compression_pointer(*label_len_octet)) {
12:         length = 2;
13:         label_len_octet += length + 1;
```

```
14:     }
15:
16:     else {
17:         length = *label_len_octet;
18:         copy(dest_octet + copy_offset,
                            label_len_octet+1, *length);
19:
20:         copy_offset += length;
21:         label_len_octet += length + 1;
22:     }
23:
24:  }
25:}
```

Snippet 4 - A typical implementation of a function that is used for copying non-compressed domain names (pseudocode)

This implementation does not explicitly check for the value of the label length octet: this value can be up to 255 octets, and a single label can fill the name buffer. Depending on the memory layout of the target, how the name buffer is allocated, and the size of the malformed packet, it is possible to trigger various memory corruption issues.

Both Snippets 1 and 4 restrict the size of the name buffer to 255 octets, however there are no restrictions on the actual number of octets that will be copied into this buffer. In this particular case, a subsequent copy operation (if another label is present in the packet) will write past the name buffer, allowing to overwrite heap or stack metadata in a controlled manner.

4. NULL-terminator Placement Validation

A domain name must end with a NULL (0x00) octet, as per [RFC1035]. The implementations shown at Snippets 1 and 4 assume that this is the case for the RRs that they process, however names that do not have a NULL octet placed at the proper position within a RR are not discarded.

This issue is closely related to the absence of label and name length checks. For example, the logic behind Snippets 1 and 4 will continue to copy octets into the name buffer, until a NULL octet is encountered. This octet can be placed at an arbitrary position within a RR, or not placed at all.

Consider a pseudocode function shown on Snippet 5. The function returns the length of a domain name ("name") in octets to be used elsewhere (e.g., to allocate a name buffer of a certain size): for compressed domain names the function returns 2, for uncompressed names it returns their true length using the "strlen()" function.

```
1: get_name_length(*name) {
```

```
2:
3:     if (is_compression_pointer(name))
4:         return 2;
5:
6:     name_len = strlen(name) + 1;
7:     return name_len;
8: }
```

Snippet 5 - A function that returns the length of a domain name

"strlen()" is a standard C library function that returns the length of a given sequence of characters terminated by the NULL (0x00) octet. Since this function also expects names to be explicitly NULL-terminated, the return value "strlen()" may be also controlled by attackers. Through the value of "name_len" attackers may control the allocation of internal buffers, or specify the number by octets copied into these buffers, or other operations depending on the implementation specifics.

The absence of explicit checks for the NULL octet placement may also facilitate controlled memory reads and writes.

5. Response Data Length Validation

As stated in [RFC1035], every RR contains a variable length string of octets that contains the retrieved resource data (RDATA) (e.g., an IP address that corresponds to a domain name in question). The length of the RDATA field is regulated by the resource data length field (RDLENGTH), that is also present in an RR.

Implementations that process RRs may not check for the validity of the RDLENGTH field value, when retrieving RDATA. Failing to do so may lead to out-of-bound read issues (similarly to the label and name length validation issues discussed in Section 3), whose impact may vary significantly depending on the implementation specifics. The authors observed instances of Denial-of-Service conditions and information leaks.

6. Record Count Validation

According to [RFC1035], the DNS header contains four two-octet fields that specify the amound of question records (QDCOUNT), answer records (ANCOUNT), authority records (NSCOUNT), and additional records (ARCOUNT).

```
1: process_dns_records(dns_header, ...) {
       // ...
2:     num_answers = dns_header->ancount
3:     data_ptr = dns_header->data
4:
5:     while (num_answers > 0) {
6:         name_length = get_name_length(data_ptr);
```

```
 7:         data_ptr += name_length + 1;
 8:
 9:         answer = (struct dns_answer_record *)data_ptr;
10:
11:         // process the answer record
12:
13:         --num_answers;
14:     }
        // ...
15: }
```

Snippet 6 - A RR processing function

Snippet 6 illustrates a recurring implementation anti-pattern for a function that processes DNS RRs. The function "process_dns_records()" extracts the value of ANCOUNT ("num_answers") and the pointer to the DNS data payload ("data_ptr"). The function processes answer records in a loop decrementing the "num_answers" value after processing each record, until the value of "num_answers" becomes zero. For simplicity, we assume that there is only one domain name per answer. Inside the loop, the code calculates the domain name length "name_length", and adjusts the data payload pointer "data_ptr" by the offset that corresponds to "name_length + 1", so that the pointer lands on the first answer record. Next, the answer record is retrieved and processed, and the "num_answers" value is decremented.

If the ANCOUNT number retrieved from the header ("dns_header->ancount") is not checked against the amount of data available in the packet and it is, e.g., larger than the number of answer records available, the data pointer "data_ptr" will read out of the bounds of the packet. This may result in Denial-of-Service conditions.

In this section, we used an example of processing answer records. However, the same logic is often reused for processing other types of records. Therefore all record count fields must be checked before parsing the contents of a packet. [RFC5625] recommends that DNS packets with wrong RR count fields should be dropped.

7. General Recommendations

7.1. Compression Pointer

A compression pointer (a byte with the 2 highest bits set to 1) must resolve to a byte within a DNS record with the value that is greater than 0 (i.e., it must not be a NULL terminator) and less than 64. The offset at which this byte is located must be smaller than the offset at which the compression pointer is located. There is no valid reason for nesting compression pointers. The code that implements domain name parsing should check the offset not only with respect to the bounds of a packet, but also its position with respect to the compression pointer in question. A compression pointer must not be

"followed" more than once. This might be difficult to implement within the logic of TCP/IP stacks, as the authors have seen several implementations using a check that ensures that a compression pointer is not followed more than several times. While this is not a perfect solution, it may still be a practical one.

7.2 Name, Label, and Resource Record Lengths

A domain name length byte must have the value of more than 0 and less than 64 ([RFC1035]). If this is not the case, an invalid value has been provided within the packet, or a value at an invalid position might be interpreted as a domain name length due to other errors in the packet (e.g., misplaced NULL terminator or invalid compression pointer). The characters of the domain label allowed for Internet hosts must strictly conform to [RFC1035], and the number of domain label characters must correspond to the value of the domain label byte. The domain name length must not be more than 255 bytes, including the size of uncompressed domain names. The NULL octet (0x00) must be present at the end of the domain name, and within the maximum name length (255 octets).

The value of the data length byte in response DNS records (RDLENGTH) must reflect the number of bytes available in the field that describes the resource (RDATA). The format of RDATA must conform to the TYPE and CLASS fields of the RR.

7.3 Resource Record Count Fields

The values of the bytes within a DNS header that reflect the number of Question (QCOUNT), Answer (ANCOUNT), Authority (NSCOUNT) and Additional (ARCOUNT) must correspond to the actual data present within the packet. The packets with with invalid RR counts must be discarded, in accordance with [RFC5625].

8. Security Considerations

Security issues are discussed throughout this memo.

9. IANA Considerations

This document introduces no new IANA considerations. Please see [RFC6895] for a complete review of the IANA considerations introduced by DNS.

10. References

10.1 Normative References

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", RFC 1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

10.2 Informative References

[COMP-DRAFT] Koch, P., "A New Scheme for the Compression of Domain Names", Internet-Draft, draft-ietf-dnsind-local-compression-05, June 1999, Work in progress, <https://tools.ietf.org/html/draft-ietf-dnsind-local-compression-05>.

[RFC5625] Bellis, R., "DNS Proxy Implementation Guidelines", RFC 5625, August 2009, <https://www.rfc-editor.org/info/rfc5625>.

[RFC6895] Eastlake 3rd, D., "Domain Name System (DNS) IANA Considerations", RFC 6895, April 2013, <https://www.rfc-editor.org/info/rfc6982>.

Acknowledgements

We would like to thank Shlomi Oberman, who has greatly contributed to the research that led to this document.

Authors' Addresses

Stanislav Dashevskyi
Forescout Technologies
John F. Kennedylaan, 2
Eindhoven, 5612AB
The Netherlands

Email: stanislav.dashevskyi@forescout.com

Daniel dos Santos
Forescout Technologies
John F. Kennedylaan, 2
Eindhoven, 5612AB
The Netherlands

Email: daniel.dossantos@forescout.com

Jos Wetzels
Forescout Technologies
John F. Kennedylaan, 2
Eindhoven, 5612AB
The Netherlands

Email: jos.wetzels@forescout.com

Amine Amri
Forescout Technologies
John F. Kennedylaan, 2
Eindhoven, 5612AB
The Netherlands

Email: amine.amri@forescout.com