Talos Vulnerability Report

TALOS-2020-0996

Videolabs libmicrodns 0.1.0 TXT record RDATA-parsing denial-of-service vulnerability

MARCH 23, 2020

CVE NUMBER

CVE-2020-6073

Summary

An exploitable denial-of-service vulnerability exists in the TXT record-parsing functionality of Videolabs libmicrodns 0.1.0. When parsing the RDATA section in a TXT record in mDNS messages, multiple integer overflows can be triggered, leading to a denial of service. An attacker can send an mDNS message to trigger this vulnerability.

Tested Versions

Videolabs libmicrodns 0.1.0

Product URLs

https://github.com/videolabs/libmicrodns

CVSSv3 Score

7.5 - CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

CWE

CWE-190: Integer Overflow or Wraparound

Details

The libmicrodns library is an mDNS resolver that aims to be simple and compatible cross-platform.

The function `mdns_recv` reads and parses an mDNS message:

```
static int
mdns_recv(const struct mdns_conn* conn, struct mdns_hdr *hdr, struct rr_entry **entries)
{
        uint8_t buf[MDNS_PKT_MAXSZ];
        size_t num_entry, n;
        ssize_t length;
        struct rr_entry *entry;

        *entries = NULL;
        if ((length = recv(conn->sock, (char *) buf, sizeof(buf), 0)) < 0)       // [1]
                return (MDNS_NETERR);

        const uint8_t *ptr = mdns_read_header(buf, length, hdr);                 // [2]
        n = length;

        num_entry = hdr->num_qn + hdr->num_ans_rr + hdr->num_add_rr;
        for (size_t i = 0; i < num_entry; ++i) {
                entry = calloc(1, sizeof(struct rr_entry));
                if (!entry)
                        goto err;
                ptr = rr_read(ptr, &n, buf, entry, i >= hdr->num_qn);            // [3]
                if (!ptr) {
                        free(entry);
                        errno = ENOSPC;
                        goto err;
                }
                entry->next = *entries;
                *entries = entry;
        }
        ...
}
```

At [1], a message is read from the network. The 12-bytes mDNS header is then parsed at [2]. Based on the header info, the loop parses each resource record (“RR”) using the function `rr_read` [3].

```
const uint8_t *
rr_read(const uint8_t *ptr, size_t *n, const uint8_t *root, struct rr_entry *entry, int8_t ans)
{
        size_t skip;
        const uint8_t *p;

        p = ptr = rr_read_RR(ptr, n, root, entry, ans);               // [4]
        if (ans == 0) return ptr;

        for (size_t i = 0; i < rr_num; ++i) {
                if (rrs[i].type == entry->type) {
                        ptr = (*rrs[i].read)(ptr, n, root, entry);    // [5]
                        if (!ptr)
                                return (NULL);
                        break;
                }
        }
        ...
```

The function `rr_read_RR` [4] reads the current resource record, except for the `RDATA` section. This is read by the loop at [5]. For each `RR` type, a different function is called. When the `RR` type is 0x10, the function `rr_read_TXT` is called at [5].

```
#define advance(x) ptr += x; *n -= x

static const uint8_t *
rr_read_TXT(const uint8_t *ptr, size_t *n, const uint8_t *root, struct rr_entry *entry)
{
        union rr_data *data = &entry->data;
        uint16_t len = entry->data_len;                         // [8]
        uint8_t l;

        if (*n == 0 || *n < len)
                return (NULL);

        for (; len > 0; len -= l + 1) {                         // [9]
                struct rr_data_txt *text;

                memcpy(&l, ptr, sizeof(l));                     // [6]
                advance(1);
                if (*n < l)
                        return (NULL);
                text = malloc(sizeof(struct rr_data_txt));
                if (!text)
                        return (NULL);
                text->next = data->TXT;
                data->TXT = text;
                if (l > 0)
                        memcpy(text->txt, ptr, l);              // [7]
                text->txt[l] = '\0';
                advance(l);                                     // [10]
        }
        return (ptr);
}
```

This function expects four parameters:

- `ptr`: the pointer to the start of the label to parse
- `n`: the number of remaining bytes in the message, starting from ptr
- `root`: the pointer to the start of the mDNS message
- `entry`: the entry struct, containing the parsed resource record

The function is supposed to extract each variable-length string from the `RDATA` section. In this case, it extracts a length in position 0 [6], and copies the data found in `text->txt` [7]. During this parsing, `*n` and `len` are decremented accordingly. In this loop, `len` tracks the number of characters left to read in the same RDATA section, as previously declared in the `data_len` field [8].

However, note that both `*n` and `len` are unsigned integers. This means that the loop will only stop when `len` is exactly equal to 0, or when `*n` is less than the length read in `RDATA`.

Also note that the `advance` macro is moving `ptr` forward and decrements `*n` (the number of bytes left in the packet) accordingly.

So, by making `l` at [6] equal to `*n`, and having at the same time `len` less than or equal to `l`, will cause `*n` to be 0 after [10], since we advance by `l`. Right after this, `len` overflows because `l` is bigger than `len`, making the loop itself cycle indefinitely. Then, a new `l` is read at [6], and `advance` is called again, making `*n` overflow. At this point both `*n` and `len` are overflown and the program will eventually crash with an out-of-bounds read at [7] or [6].

Timeline

2020-01-30 - Vendor Disclosure
2020-03-20 - Vendor Patched
2020-03-23 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.
