Talos Vulnerability Report

TALOS-2020-1002

Videolabs libmicrodns 0.1.0 resource allocation denial-of-service vulnerabilities

MARCH 23, 2020

CVE NUMBER

CVE-2020-6079, CVE-2020-6080

Summary

Multiple exploitable denial-of-service vulnerabilities exist in the resource allocation handling of Videolabs libmicrodns 0.1.0. When encountering errors while parsing mDNS messages, some allocated data is not freed, possibly leading to a denial-of-service condition via resource exhaustion. An attacker can send one mDNS message repeatedly to trigger these vulnerabilities.

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-400: Uncontrolled Resource Consumption (‘Resource Exhaustion’)

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);                                             // [4]
                        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);               // [5]
        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);    // [6]
                        if (!ptr)
                                return (NULL);                        // [7]
                        break;
                }
        }
        ...
```

CVE-2020-6079 - rr_decode

The function `rr_read`, in turn calls `rr_read_RR` [5]:

```
static const uint8_t *
rr_read_RR(const uint8_t *ptr, size_t *n, const uint8_t *root, struct rr_entry *entry, int8_t ans)
{
        uint16_t tmp;

        ptr = rr_decode(ptr, n, root, &entry->name);
        if (!ptr || *n < 4)
                return (NULL);                                        // [8]

        ptr = read_u16(ptr, n, &entry->type);
        ptr = read_u16(ptr, n, &tmp);
        entry->rr_class = (tmp & ~0x8000);
        entry->msbit = ((tmp & 0x8000) == 0x8000);
        if (ans) {
                if (*n < 6)
                        return (NULL);                                // [9]
                ptr = read_u32(ptr, n, &entry->ttl);
                ptr = read_u16(ptr, n, &entry->data_len);
        }
        return ptr;
}
```

The actual decoding of the domain name is performed by `rr_decode`:

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

/*
 * Decodes a DN compressed format (RFC 1035)
 * e.g "\x03foo\x03bar\x00" gives "foo.bar"
 */
static const uint8_t *
rr_decode(const uint8_t *ptr, size_t *n, const uint8_t *root, char **ss)
{
        char *s;

        s = *ss = malloc(MDNS_DN_MAXSZ);                              // [10]
        if (!s)
                return (NULL);

        if (*ptr == 0) {
                *s = '\0';
                advance(1);
                return (ptr);
        }
        ...
        advance(1);
        return (ptr);
err:                                                                  // [11]
        free(*ss);
        return (NULL);
}
```

The function `rr_decode` allocates the `ss` buffer [10], which is only freed upon error [11]. This means that the caller of this function is responsible for free-ing this buffer.

We can see that, if the conditions at [8] or [9] are hit, the code would return NULL without free-ing the `entry->name` buffer (called `ss` in `rr_decode`). Eventually, `mdns_recv` will free the structure `entry` [4], but will not try to free anything inside it. Note however, that due to a bug discussed in TALOS-2020-1000, these conditions are not reachable.

However, there is another opportunity to trigger this bug later, at [7]. Inside that loop, for each `RR` type, a different function is called. So, to trigger the `return NULL` at [7] an attacker could specify a message with an invalid `SRV`, `PTR`, `TXT`, `AAAA`, `A` structure, in order to make any of those functions to fail and return NULL.

CVE-2020-6080 - `rr_read_TXT`

The function `rr_read_RR` [5] reads the current resource record, except for the `RDATA` section. This is read by the loop at in `rr_read`. For each `RR` type, a different function is called. When the `RR` type is 0x10, the function `rr_read_TXT` is called at [6].

```
#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;                         // [15]
        uint8_t l;

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

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

                memcpy(&l, ptr, sizeof(l));                     // [12]
                advance(1);
                if (*n < l)                                     // [16]
                        return (NULL);
                text = malloc(sizeof(struct rr_data_txt));      // [14]
                if (!text)
                        return (NULL);
                text->next = data->TXT;
                data->TXT = text;
                if (l > 0)
                        memcpy(text->txt, ptr, l);              // [13]
                text->txt[l] = '\0';
                advance(l);
        }
        return (ptr);
}
```

This function expects 4 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 [12], and copies the data found in `text->txt` [13], after allocating space for it at [14]. 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 [15].

Note that, because of the loop, the code would parse multiple strings in the same `RDATA` section. However, if the condition at [16] is met, the function returns `NULL` (which suggests the caller function to discard the record altogether) without first free-ing the allocated `text` structures.

Thus, any TXT answer with more than one string in the `RDATA` section, when also containing an invalid string length at the end, would trigger the condition at [16], causing a resource leak. An attacker can exploit this behavior by sending multiple TXT answers, exhausting the process memory and crashing the service.

Timeline

2020-01-30 - Vendor Disclosure

2020-03-20 - Vendor Patched

2020-03-23 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT

NEXT REPORT

TALOS-2020-1001

TALOS-2020-1039

© 2022 Cisco Systems, Inc. and/or its affiliates. All rights reserved. View our Privacy Policy.