Talos Vulnerability Report

TALOS-2020-1001

Videolabs libmicrodns 0.1.0 mdns_recv return value denial-of-service vulnerability

MARCH 23, 2020

CVE NUMBER

CVE-2020-6078

Summary

An exploitable denial-of-service vulnerability exists in the message-parsing functionality of Videolabs libmicrodns 0.1.0. When parsing mDNS messages in `mdns_recv`, the return value of the `mdns_read_header` function is not checked, leading to an uninitialized variable usage that eventually results in a null pointer dereference, leading to service crash. An attacker can send a series of mDNS messages 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-252: Unchecked Return Value

Details

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

The function `mdns_listen_probe_network` handles the data structures used for the reception of mDNS messages. It declares an `mdns_hdr` that stores the header of the last mDNS message processed. It also calls `mdns_recv` [2], which actually fills this structure and parses the rest of the mDNS message. As we can see, the structure is initialized to `0`, but it's left untouched inside the `mdns_recv` loop.

```
struct mdns_hdr {
        uint16_t id;
        uint16_t flags;
        uint16_t num_qn;
        uint16_t num_ans_rr;
        uint16_t num_auth_rr;
        uint16_t num_add_rr;
};

...

static int
mdns_listen_probe_network(const struct mdns_ctx *ctx, const char *const names[],
                          unsigned int nb_names, mdns_listen_callback callback,
                          void *p_cookie)
{
    struct mdns_hdr ahdr = {0};                                                 // [1]
    struct rr_entry *entries;
    struct pollfd *pfd = alloca( sizeof(*pfd) * ctx->nb_conns );
    int r;

    for (size_t i = 0; i < ctx->nb_conns; ++i) {
            pfd[i].fd = ctx->conns[i].sock;
            pfd[i].events = POLLIN;
    }

    r = poll(pfd, ctx->nb_conns, 1000);
    if (r <= 0) {
            return r;
    }
    for (size_t i = 0; i < ctx->nb_conns; ++i) {
            if ((pfd[i].revents & POLLIN) == 0)
                    continue;
            r = mdns_recv(&ctx->conns[i], &ahdr, &entries);                     // [2]
            if (r == MDNS_NETERR && os_wouldblock())
            {
                    mdns_free(entries);
                    continue;
            }

            if (ahdr.num_ans_rr + ahdr.num_add_rr == 0)
            {
                    mdns_free(entries);
                    continue;
            }

            for (struct rr_entry *entry = entries; entry; entry = entry->next) {
                    for (unsigned int i = 0; i < nb_names; ++i) {
                            if (!strrcmp(entry->name, names[i])) {
                                    callback(p_cookie, r, entries);
                                    break;
                            }
                    }
            }
            mdns_free(entries);
    }
    return 0;
}
```

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)      // [3]
                return (MDNS_NETERR);

        const uint8_t *ptr = mdns_read_header(buf, length, hdr);                // [4]
        n = length;

        num_entry = hdr->num_qn + hdr->num_ans_rr + hdr->num_add_rr;            // [5]
        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);           // [6]
                if (!ptr) {
                        free(entry);
                        errno = ENOSPC;
                        goto err;
                }
                entry->next = *entries;
                *entries = entry;
        }
        ...
}
```

At [3], a message is read from the network. The 12-bytes mDNS header is then parsed at [4]. If at least one question/resource-record is found [5], the loop parses the remaining data in the message. by calling `rr_read` [6].

```
static const uint8_t *
mdns_read_header(const uint8_t *ptr, size_t n, struct mdns_hdr *hdr)
{
        if (n <= sizeof(struct mdns_hdr)) {
                errno = ENOSPC;
                return NULL;                                // [7]
        }
        ptr = read_u16(ptr, &n, &hdr->id);
        ptr = read_u16(ptr, &n, &hdr->flags);
        ptr = read_u16(ptr, &n, &hdr->num_qn);
        ptr = read_u16(ptr, &n, &hdr->num_ans_rr);
        ptr = read_u16(ptr, &n, &hdr->num_auth_rr);
        ptr = read_u16(ptr, &n, &hdr->num_add_rr);
        return ptr;
}
```

The function `mdns_read_header` parses the header, and returns NULL [7] when the message is too small (less than or equal to 12). However, after [4], the code doesn't check the return value of this function. Since the contents of the `hdr` structures are not reset between each call of `mdns_recv`, the line at [5] is effectively accessing uninitialized data.
If `num_entry` is then different from `0` (because of a previous valid mDNS message), the `rr_read` function will be called with `ptr` set to `NULL`. Eventually, `rr_read` will call the `rr_decode` function that will dereference this null pointer at [8], crashing the service.

```
/*
 * 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);
        if (!s)
                return (NULL);

        if (*ptr == 0) {                                            // [8]
                *s = '\0';
                advance(1);
                return (ptr);
        }
        ...
```

Timeline

2020-01-30 - Vendor Disclosure

2020-03-20 - Vendor Patched

2020-03-23 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.
