Talos Vulnerability Report

TALOS-2022-1479

Anker Eufy Homebase 2 libxm_av.so getpeermac() authentication bypass vulnerability

MAY 5, 2022

CVE NUMBER

CVE-2022-25989

SUMMARY

An authentication bypass vulnerability exists in the libxm_av.so getpeermac() functionality of Anker Eufy Homebase 2 2.1.8.5h. A specially-crafted DHCP packet can lead to authentication bypass. An attacker can DHCP poison to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

Anker Eufy Homebase 2 2.1.8.5h

PRODUCT URLS

Eufy Homebase 2 - https://us.eufylife.com/products/t88411d1

CVSSV3 SCORE

7.1 - CVSS:3.0/AV:A/AC:L/PR:N/UI:N/S:C/C:L/I:L/A:L

CWE

CWE-290 - Authentication Bypass by Spoofing

DETAILS

The Eufy Homebase 2 is the video storage and networking gateway that enables the functionality of the Eufy Smarthome ecosystem. All Eufy devices connect back to this device, and this device connects out to the cloud, while also providing assorted services to enhance other Eufy Smarthome devices.

Among the `home_security` binary's responsibilities, communications with the cloud and with smarthome devices is the most important. While the binary itself is somewhat opaque with regards to the actual implementation of this, a good chunk of the network functionality is within an imported `libxm_av.so` library. This library normally creates five different network servers, as so:

```
tcp
  32392 - UDPRecvClient path
  32293 - WifiComSend_Pth
  32295 - WifiComRecv_Pth
  32290 - DspComSvr_Path - recv   ** not seen **
  32292 - DspComSvr_Path - send   ** not seen **

udp
  32380 - UDPComCreate => UdpRecvSvr_pth
  32392 - UdpSndSvr
```

While the code paths of some of these seem to converge or complement each other, these servers are all related to communications between the Homebase 2 and the smarthome devices. These communications occur over a seperate network than that of the Homebase's normal ethernet connection, as it broadcasts a WiFi Hotspot with a hidden SSID that all the smarthome devices can connect to. This WiFi hotspot corresponds to the `br0` interface in the Homebase:

```
8: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 8c:85:80:AA:BB:CC brd ff:ff:ff:ff:ff:ff
    inet 192.168.32.2/24 brd 192.168.32.255 scope global br0
       valid_lft forever preferred_lft forever
    inet6 fe80::8e85:80ff:feaa:bbcc/64 scope link
       valid_lft forever preferred_lft forever
```

When a device is paired to the Homebase, it is statically assigned an IP address in the range of `192.168.32.5` to `192.168.32.21`, but if one manually connects to this WiFi AP, a DHCP IP address is assigned starting from `192.168.32.100`. While this might seem irrelevant, there are indeed hardcoded checks within these servers to see whether a connected client falls within the `192.168.32.5-192.168.32.21` IP address range, which could determine if a device is authorized to use an opcode or not. There also exist comparisons to the MAC addresses of

paired devices, as another authentication method. But for today's vulnerability, we examine the somewhat related `getpeermac()` function, which is used to check whether or not a client to any of the above mentioned TCP/UDP ports is allowed to send traffic:

```
int32_t getpeermac(int32_t fd, char* sprintf_out){

    int32_t addrlne = 0x10
    struct arpreq arp
    memset(addr: &arp, chr: 0, size: 0x44)

    struct sockaddr addr
    addr.sin_family.d = 0
    addr.inet_addr = 0
    addr.pad[0].d = 0
    addr.pad[4].d = 0

    int32_t peerret = getpeername(fd, &addr, &addrlne) // [1]
    uint32_t $a1_1 = 0xdd
    char* fmtstr
    int32_t $v0
    if (peerret s< 0)
        fmtstr = "getpeermac: getpeername err\n"
    else
        arp.arp_dev[0].d = 'br0'
        arp.arp_pa.sin_family.d = addr.sin_family.d
        arp.arp_pa.inet_addr = addr.inet_addr // [2]
        arp.arp_pa.pad[0].d = addr.pad[0].d
        arp.arp_pa.pad[4].d = addr.pad[4].d
        arp.arp_pa.sin_family = 2
        arp.arp_ha.sin_family = 0
        iret = ioctl(fd, 0x8954, &arp)        // [3]
        if (iret s< 0)
            fmtstr = "getpeermac: ioctrl err\n"
            $a1_1 = 0xe7
    int32_t ret
    if (peerret s< 0 || (peerret s>= 0 && iret s< 0))
        uint8_t* var_7c
        XM_LOG(fname: "PrivateAPI.c", size: $a1_1, 5, 0, fmtstr: fmtstr, values:
var_7c)
        ret = 0xfffff447
    if (peerret s>= 0 && $v0 s>= 0)
        sprintf(sprintf_out, 0x3d1d0, zx.d(arp.arp_ha.sin_port.b), [...]
{"%02X:%02X:%02X:%02X:%02X:%02X"} // [4]
        ret = 0
    return ret
}
```

To summarize, at [1], the function gets the client socket IP address of our connection, placing it into `addr`. At [2], this `addr`'s data is put into the `arpreq` struct. At [3], the 0x8954 ioctl is done, which asks the kernel if there are any arp table entries on the `br0` interface that have an IP address corresponding to our client socket. Assuming this all

passes, a MAC address string is populated into the `sprintf_out` parameter pointer at [4], and 0x0 is returned by this function. If `getpeermac` does not return 0x0, the connection is immediately terminated, resulting in an authentication mechanism that is determined by the connection's interface. To put more plainly, if we try sending traffic to any of these ports over the Homebase's ethernet connection, the connection is blocked. Only paired smarthome devices are allowed to talk to these servers.

There is however an oversight in this implementation, namely that the Homebase's `eth0` connection is DHCP. Thus, if an attacker is on the same subnet as the Homebase and can DHCP poison the Homebase, the `192.168.32.0/24` range can be placed on both the `eth0` interface and the `br0` interface. If the attacker subsequently assigns themselves an IP address that matches any device on the `br0` interface, then the call to `getpeermac()` will actually succeed. Again looking at the important parts of `getpeermac()`:

```
    int32_t peerret = getpeername(fd, &addr, &addrlne)
```

If our IP address legitimately is `192.168.32.5` for instance, the same IP address as a smarthome device on `br0`, then `getpeername` will populate `addr` with `192.168.32.5 (0xc0a82005)`.

```
   iret = ioctl(fd, 0x8954, &arp)
```

Since we're looking up `192.168.32.5`: Assuming there is an actual device with this IP address, the `ioctl` will return successfully with the MAC address of the device on `br0`, allowing us to talk to these ports. But there is one more step to take in this setup. If we put the `192.168.32.0/24` network on the `eth0` interface, we actually won't see any return traffic due to routing priority. To solve this issue, we can simply poison instead with `192.168.32.0/25` or any other smaller subnet that lets us share an IP address with a device on the `br0` interface. Since smaller subnets by default take priority to larger subnets in routing tables, we can force the Homebase to send traffic to us instead of the Smarthome devices on `br0`, regardless of the static arp table entries that get configured for paired devices.

VENDOR RESPONSE

Fixed version 3.1.8.7 and 3.1.8.7h is in grayscale on Homebase2

TIMELINE

2022-03-11 - Vendor disclosure

2022-04-15 - Vendor patched

2022-05-05 - Public disclosure

CREDIT

Discovered by Lilith >_> of Cisco Talos.

VULNERABILITY REPORTS PREVIOUS REPORT NEXT REPORT

TALOS-2022-1465 TALOS-2022-1480