Hacking the Supply Chain

The Ripple20 Vulnerabilities Haunt Hundreds of Millions of Critical Devices

DEF CON 28 Safe Mode
Who are we?

**JSOF** is a software security consultancy

- **Shlomi Oberman**, co-founder, JSOF
- **Moshe Kol**, Security researcher, JSOF; Finder of Ripple20
- **Ariel Schön**, Security researcher, JSOF
Agenda

• Ripple20

• CVE-2020-11901

• Exploiting CVE-2020-11901
Ripple20

• Series of 19 zero-day vulnerabilities in Treck TCP/IP*

• Amplified by the supply chain

• 100’s of millions of devices

• Medical, ICS, Home, Enterprise, Transportation, Utilities
### Ripple20

<table>
<thead>
<tr>
<th>CVE-2020-11896</th>
<th>CVE-2020-11901</th>
<th>CVE-2020-11906</th>
<th>CVE-2020-11911</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2020-11897</td>
<td>CVE-2020-11902</td>
<td>CVE-2020-11907</td>
<td>CVE-2020-11912</td>
</tr>
<tr>
<td>CVE-2020-11898</td>
<td>CVE-2020-11903</td>
<td>CVE-2020-11908</td>
<td>CVE-2020-11913</td>
</tr>
<tr>
<td>CVE-2020-11899</td>
<td>CVE-2020-11904</td>
<td>CVE-2020-11909</td>
<td>CVE-2020-11914</td>
</tr>
<tr>
<td>CVE-2020-11900</td>
<td>CVE-2020-11905</td>
<td>CVE-2020-11910</td>
<td></td>
</tr>
</tbody>
</table>

- 4 critical remote code execution vulnerabilities
Ripple20

- CVE-2020-11896
- CVE-2020-11897
- CVE-2020-11898
- CVE-2020-11899
- CVE-2020-11900
- CVE-2020-11901
- CVE-2020-11902
- CVE-2020-11903
- CVE-2020-11904
- CVE-2020-11905
- CVE-2020-11906
- CVE-2020-11907
- CVE-2020-11908
- CVE-2020-11909
- CVE-2020-11910
- CVE-2020-11911
- CVE-2020-11912
- CVE-2020-11913
- CVE-2020-11914

- 8 medium-high severity vulnerabilities
100’s of Millions of Devices Affected

And many more...
100’s of Millions of Devices Affected

- Medical
- Printers
- Utilities
- Transportation
- Networking
- Datacenter
- Smart Buildings
- Industrial

• Assumption: Every mid-large US organization has one
Supply chain

1. Library (Source)
2. Operating System
3. System on Module
4. IV Pump
Supply chain

1. Library (Source)
2. Operating System
3. System on Module
4. IV Pump
Vulnerabilities
Why Treck TCP/IP?

• Supply chain - mostly unexplored
• 1 vulnerability == multiple products
• Large IoT impact
• Zombie vulnerabilities
• Good attack surface
Trekk TCP/IP

- Treck is a small American company
- Treck TCP/IP is a proprietary TCP/IP stack; Available >20 years
- Embedded devices and RTOS
- Very configurable. Each Treck instance is different.
- Strategically located at the start of a long supply-chain
Ripple20 Research

• Reverse engineering of 6 different devices with multiple versions

• Every device has a different configuration

• Ongoing research Sep’19 - Jun’20 ( 9 months )

• Some strange architectures and firmwares involved

2 whitepapers released
About CVE-2020-11901

• Critical vulnerabilities in Treck’s DNS Resolver component.
• Once successfully exploited, allows for remote code execution.
• Can traverse NAT boundaries.

• 4 vulnerabilities and 1 artifacts.
• Vary over time and vendor.
CVE-2020-11901

AKA “the DNS bugs”
DNS Primer: The Basics

- The DNS protocol maps between **domain names** and **IP addresses**.
- Client **resolves** a name by issuing a query to a DNS server.
- The DNS server **looks up** the name and returns a response.
DNS Primer: Record Types

• DNS servers can return multiple answers in the same DNS response.
• An answer is specified as a resource record:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLASS</th>
<th>TTL</th>
<th>RDLENGTH</th>
<th>RDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(var)</td>
<td>(2 bytes)</td>
<td>(2 bytes)</td>
<td>(4 bytes)</td>
<td>(2 bytes)</td>
<td>(var)</td>
</tr>
</tbody>
</table>

• Questions and answers have a type. Common types include:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>IPv4 address for the queried domain.</td>
</tr>
<tr>
<td>CNAME</td>
<td>Alias (canonical name).</td>
</tr>
<tr>
<td>MX</td>
<td>Domain name of a mail server for the queried domain.</td>
</tr>
</tbody>
</table>
Domain Names Encoding

• Domain names are encoded as a sequence of labels.
• Each label is preceded by a length byte.
• Maximum label length is 63.
DNS Message Compression

• Compression is achieved by replacing a sequence of labels with a **pointer to prior occurrence** of the same sequence.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>0xabcd</td>
<td>0x8180</td>
<td>0x0001</td>
<td>0x0001</td>
<td>0x0000</td>
<td>0x0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0xc</td>
<td>5</td>
<td>g</td>
<td>m</td>
<td>a</td>
<td>i</td>
<td>l</td>
<td>3</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>0</td>
</tr>
<tr>
<td>+0x18</td>
<td>0x0f</td>
<td>0x0001</td>
<td>0xc0</td>
<td>0xc0</td>
<td>0x000f</td>
<td>0x0001</td>
<td>0x000151</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x24</td>
<td>0x80</td>
<td>0x0009</td>
<td>0x0000</td>
<td>4</td>
<td>s</td>
<td>m</td>
<td>t</td>
<td>p</td>
<td>0xc0</td>
<td>0xc0</td>
<td></td>
</tr>
</tbody>
</table>

• Compression pointer is encoded in **two bytes**, the first begins with 11.
DNS Parsing Logic: Type MX

```c
if (cacheEntryQueryType == DNS_TYPE_MX && rrtype == DNS_TYPE_MX) {
    addr_info = tfDnsAllocAddrInfo();
    if (addr_info != NULL) {
        /* copy preference value of MX record */
        memcpy(&addr_info->ai_mxpref, resourceRecordAfterNamePtr + 10, 2);

        /* compute the length of the MX hostname */
        labelLength = tfDnsExpLabellength(resourceRecordAfterNamePtr + 0xc, pktDataPtr);
        addr_info->ai_mxhostname = NULL;
        if (labelLength != 0) {
            /* allocate buffer for the expanded name */
            asciiPtr = tfGetRawBuffer(labelLength);
            addr_info->ai_mxhostname = asciiPtr;
            if (asciiPtr != NULL) {
                /* copy MX hostname to `asciiPtr` as ASCII */
                tfDnsLabelToAscii(resourceRecordAfterNamePtr + 0xc, asciiPtr, pktDataPtr);
                /* ... */
            }
        }
    }
}
```
DNS Label Length Calculation

```cpp
tt16Bit tfDnsExpLabelLength(tt8BitPtr labelPtr, tt8BitPtr pktDataPtr){
    tt8Bit currLabelLength;
    tt16Bit i = 0, totalLength = 0;

    while (labelPtr[i] != 0) {
        currLabelLength = labelPtr[i];
        if ((currLabelLength & 0xc0) == 0) {
            totalLength += currLabelLength + 1;
            i += currLabelLength + 1;
        } else {
            newLabelPtr = pktDataPtr + (((currLabelLength & 0x3f) << 8) | labelPtr[i+1]);
            if (newLabelPtr >= labelPtr) {
                return 0;
            }
        }
    }
    labelPtr = newLabelPtr;
    i = 0;
}
return totalLength;
```

- Reads the current label length
- Handles the common case: no compression
- Reads the compression offset
- Only allows jumping backwards
Vulnerability #1: Read Out-Of-Bounds

- `tfDnsExpLabelLength` might read data out of the packet buffer while iterating over the length bytes (stops at a zero length byte).
- Could result in **denial-of-service** (e.g., read from unmapped page).

**Information leakage:**
- `tfDnsLabelToAscii` has no bounds check either.
- Data from the heap could be interpreted as an MX hostname.
- Data is leaked when the client tries to resolve the MX hostname.
- Affects Treck version 4.7+, fixed later.
- Sweet! but we want RCE...
More Issues with tfDnsExpLabelLength

- Maximum domain name of 255 characters is not enforced.
- Does not validate the characters of the domain name: should be alphanumeric and ‘-’ only.
- `totalLength` variable is stored as an **unsigned short** (tt16Bit).

```c
tt16Bit tfDnsExpLabelLength(tt8BitPtr labelPtr, tt8BitPtr pktDataPtr){
  tt8Bit currLabelLength;
  tt16Bit i = 0, totalLength = 0;
  /* ... */
  return totalLength;
}
```
More Issues with tfDnsExpLabelLength

• Maximum domain name of 255 characters is not enforced.
• Does not validate the characters of the domain name: should be alphanumeric and ‘-’ only.
• `totalLength` variable is stored as an unsigned short (`tt16Bit`).
Vulnerability #2: Integer Overflow

• We need to construct a name whose length is larger than 65536.
• **Can we overflow the totalLength variable within a DNS response packet?**
• **Yes!** We use the DNS compression feature to achieve this.
• Idea: nested compression pointers.
• Two challenges:
  • Maximum size of the DNS response packet allowed is 1460 bytes.
  • We can only jump **backwards** from our current label pointer.
Vulnerability #2: Integer Overflow

<table>
<thead>
<tr>
<th>+0</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
<th>0f</th>
</tr>
</thead>
<tbody>
<tr>
<td>+16</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+32</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+48</td>
<td>00</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+64</td>
<td>c0</td>
<td>00</td>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+80</td>
<td>c0</td>
<td>01</td>
<td>c0</td>
<td>02</td>
<td>0b</td>
<td>0c</td>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+96</td>
<td>c0</td>
<td>03</td>
<td>c0</td>
<td>04</td>
<td>c0</td>
<td>05</td>
<td>c0</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0a</td>
<td>0b</td>
<td>0c</td>
</tr>
<tr>
<td>+112</td>
<td>c0</td>
<td>07</td>
<td>c0</td>
<td>08</td>
<td>c0</td>
<td>09</td>
<td>c0</td>
<td>0a</td>
<td>c0</td>
<td>0b</td>
<td>c0</td>
<td>0c</td>
<td>c0</td>
<td>0d</td>
</tr>
</tbody>
</table>

branch byte  compression pointer

totalLength = 0
Vulnerability #2: Integer Overflow

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+16</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+32</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+48</td>
<td>00</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+64</td>
<td>c0</td>
<td>00</td>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+80</td>
<td>c0</td>
<td>01</td>
<td>c0</td>
<td>02</td>
<td>0b</td>
<td>0c</td>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>+96</td>
<td>c0</td>
<td>03</td>
<td>c0</td>
<td>04</td>
<td>c0</td>
<td>05</td>
<td>c0</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0a</td>
<td>0b</td>
<td>0c</td>
<td>0d</td>
<td>0e</td>
</tr>
<tr>
<td>+112</td>
<td>c0</td>
<td>07</td>
<td>c0</td>
<td>08</td>
<td>c0</td>
<td>09</td>
<td>c0</td>
<td>0a</td>
<td>c0</td>
<td>0b</td>
<td>c0</td>
<td>0c</td>
<td>c0</td>
<td>0d</td>
<td>c0</td>
<td>0e</td>
</tr>
</tbody>
</table>

branch byte
compression pointer

totalLength=0x111
Vulnerability #2: Integer Overflow

<table>
<thead>
<tr>
<th>+0</th>
<th>+16</th>
<th>+32</th>
<th>+48</th>
<th>+64</th>
<th>+80</th>
<th>+96</th>
<th>+112</th>
</tr>
</thead>
<tbody>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>00</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>00</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>c0</td>
<td>00</td>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>c0</td>
<td>01</td>
<td>c0</td>
<td>02</td>
<td>0b</td>
<td>0c</td>
<td>0d</td>
<td>0e</td>
</tr>
<tr>
<td>c0</td>
<td>03</td>
<td>c0</td>
<td>04</td>
<td>c0</td>
<td>05</td>
<td>c0</td>
<td>06</td>
</tr>
<tr>
<td>c0</td>
<td>07</td>
<td>c0</td>
<td>08</td>
<td>c0</td>
<td>09</td>
<td>c0</td>
<td>0a</td>
</tr>
<tr>
<td>c0</td>
<td>0b</td>
<td>c0</td>
<td>0c</td>
<td>c0</td>
<td>0d</td>
<td>c0</td>
<td>0e</td>
</tr>
<tr>
<td>c0</td>
<td>0d</td>
<td>c0</td>
<td>0e</td>
<td>c0</td>
<td>0f</td>
<td>c0</td>
<td>0f</td>
</tr>
</tbody>
</table>

branch byte
compression pointer

totalLength = 1502
Vulnerability #2: Integer Overflow

• To maximize the totalLength, we used the maximum label length 63 ($0x3f$) instead of $0x0f$ shown in the example.
• Using this construction, we reached a name of length ~72700 bytes, overflowing the totalLength variable.
• We have an RCE candidate 😊
• Can be triggered in response to every query type supported - using CNAME records.
• Affects Treck versions <= 6.0.1.66.

Fast forward to the future…
Bad Fix

Bad Fix for the Read Out-Of-Bounds Vulnerability
Fixing the Read Out-Of-Bounds

```c
if (RDLENGTH <= remaining_size) {
    labelEndPtr = resourceRecordAfterNamePtr + 10 + RDLENGTH;
    if (cacheEntryQueryType == DNS_TYPE_MX && rrtype == DNS_TYPE_MX) {
        addr_info = tfDnsAllocAddrInfo();
        if (addr_info != NULL && RDLENGTH >= 2) {
            /* copy preference value of MX record */
            memcpy(&addr_info->ai_mxpref, resourceRecordAfterNamePtr + 10, 2);
            /* compute the length of the MX hostname */
            labelLength = tfDnsExpLabelLength(resourceRecordAfterNamePtr+0xc, dnsHeaderPtr, labelEndPtr);
            if (labelLength != 0) {
                /* allocate buffer for the expanded name */
                asciiPtr = tfGetRawBuffer(labelLength);
                addr_info->ai_mxhostname = asciiPtr;
                if (asciiPtr != NULL) {
                    /* copy MX hostname to `asciiPtr` as ASCII */
                    tfDnsLabelToAscii(resourceRecordAfterNamePtr + 0xc, asciiPtr, dnsHeaderPtr, 1, 0);
                }
            }
        }
    }
}
```

When `tfDnsExpLabelLength` reaches `1labelEndPtr`, it stops processing (w/o error) and returns the current `totalLength`. 

*Pseudo-code*
Vulnerability #3: Bad RDLENGTH

- `labelEndPtr` is calculated based on the RDLENGTH field of the current resource record.
- RDLENGTH is attacker-controlled! Oops…

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLASS</th>
<th>TTL</th>
<th>RDLENGTH</th>
<th>RDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>example.com</td>
<td>MX</td>
<td>IN</td>
<td>86400</td>
<td>20 7</td>
<td>0 0 4 smtp 7 example3 com 0</td>
</tr>
</tbody>
</table>

- `tfDnsExpLabelLength` returns 5;
- `tfDnsLabelToAscii` will copy the entire MX hostname.
if (RDLENGTH <= remaining_size) {
labelEndPtr = resourceRecordAfterNamePtr + 10 + RDLENGTH;
if (cacheEntryQueryType == DNS_TYPE_MX && rrtype == DNS_TYPE_MX) {
    addr_info = tfDnsAllocAddrInfo();
    if (addr_info != NULL && RDLENGTH >= 2) {
        /* copy preference value of MX record */
        memcpy(&addr_info->ai_mxpref, resourceRecordAfterNamePtr + 10, 2);
        /* compute the length of the MX hostname */
        labelLength = tfDnsExpLabelLength(resourceRecordAfterNamePtr + 0xc, dnsHeaderPtr, labelEndPtr);
        addr_info->ai_mxhostname = NULL;
        if (labelLength != 0) {
            /* allocate buffer for the expanded name */
            asciiPtr = tfGetRawBuffer(labelLength);
            addr_info->ai_mxhostname = asciiPtr;
            if (asciiPtr != NULL) {
                /* copy MX hostname to `asciiPtr` as ASCII */
                tfDnsLabelToAscii(resourceRecordAfterNamePtr + 0xc, asciiPtr, dnsHeaderPtr, 1, 0);
            }
        }
    }
}
/* ... */
}
Artifact: Memory Leak

- An `addrinfo` structure can be leaked during MX parsing logic.
- Size of the leak 0x3c.
- Comes in handy when exploiting heap vulnerabilities.
## CVE-2020-11901: Summary

<table>
<thead>
<tr>
<th>Treck Version</th>
<th>Vuln #1: Read OOB</th>
<th>Vuln #2: Integer Overflow</th>
<th>Vuln #3: Bad RDLENGTH</th>
<th>Artifact: Memory Leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>New</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Affected**
- **Not affected**

A device can be affected by one or more vulnerabilities depending on the exact version.
Exploitation

Exploiting CVE-2020-11901 on Schneider Electric UPS Device
Target Device

• Schneider Electric APC UPS network card

• Turbo186 (x86-based)
  • 16-bit Real Mode
  • No ASLR or DEP
  • Weird segmentation (shift 8 instead of 4)

• No debugging capabilities
  • Only limited crashdumps
Vulnerability Recap

- **Primitive**: heap overflow via DNS response parsing
  - Only alpha-numeric characters are copied*
- We will exploit using “bad RDLENGTH” (#3)

<table>
<thead>
<tr>
<th>Treck Version</th>
<th>Vuln #1: Read OOB</th>
<th>Vuln #2: Integer Overflow</th>
<th>Vuln #3: Bad RDLENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>New</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Exploiting Heap Overflows

- Metadata corruption
  - Free-list pointers, block sizes, etc.
- Application-specific data structures

- Metadata exploitation considered more generic
  - As demonstrated in our exploit of CVE-2020-11896

- Can we use the same technique here?
Trekk Heap (in this case...)

- Heap structure slightly different this time
- Tightest fit favored
- Adjacent free blocks are coalesced
- Free-list checked on every heap operation
  - Pre and post sizes verified against each other
- Allocated blocks checked only when free()’d
- Avoiding a premature crash with alpha-numeric overflow is hard...

Free List Block

- Size: 4 bytes
- Next: 4 bytes
- Data
- Size: 4 bytes
Exploitation Technique

• We can overflow through all DNS response types
• When the device boots*, 3 MX requests are transmitted
• **Interactivity** in exploits is advantageous
  • Allows easier shaping
• Crashing is favorable in order to reach **deterministic state**
  • No penalty* for crashing the network card
## Overflow Target

- **tsDnsCacheEntry**
  - Contains a list of `addrinfo` structs
  - `addrinfo` holds the contents of a DNS answer (name, IP address, ...)

- Has many pointers and interesting fields
- Many references in DNS response parsing
CNAME Processing

```c
if (found_cname) {
    // Get the first addrinfo struct from `tsDnsCacheEntry`
    first_addr_info = t_dns_cache_entry->dnscAddrInfoPtr;
    if (first_addr_info) {
        // get CNAME name length from the packet
        length = tfDnsExpLabelLength(cname_rdata_ptr, packet_ptr, cname_rdata_end_ptr);
        if (length) {
            // allocate
            cname_label_buffer = tfGetRawBuffer(length);
            if (cname_label_buffer) {
                // copy to new buffer
                tfDnsLabelToAscii(cname_rdata_ptr, cname_label_buffer, packet_ptr, 1, 0);
                first_addr_info->ai_canonname = cname_label_buffer;
            }
        }
    }
}
```

*Pseudo-code*
Controlled Pointer Write

- We can write a 4-byte pointer
  - (Offset, Segment)
- To any alpha-numeric address
- Relatively strong exploitation primitive
Linear Overflow

• Overflow is from end of MX name buffer
Linear Overflow

• Overflow is from end of MX name buffer
• tsDnsCacheEntry must be placed after MX name buffer
Heap Shaping - Limitations

• Overflow target: tsDnsCacheEntry
  • Allocated on DNS request creation

• Overflow source: MX name buffer
  • Allocated on DNS response parsing

• Corrupting free blocks will result in a crash
  • We must overwrite only allocated data
Heap Shaping – Target Shape

• A specific hole pattern would allow us to overflow tsDnsCacheEntry
  • Because of **tight-fit preference**

| Head ... | Hole #1 => MX name buffer | separator | Hole #2 => tsDnsCacheEntry | separator | ... Tail |

• Allocation primitives required to attain this shape
Temporary Allocation

• Every DNS answer that contains a name (MX, PTR, CNAME) causes allocation
  • Controlled size, controlled contents
  • All answer types (except CNAME) get a new addrinfo as well

• This allocation is free()’d after DNS parsing fails
  • Or DNS TTL expires

• Good for creating arbitrary sized free regions
// first, check if the type fits for MX parsing
if (t_dns_cache_entry.dnscQueryType == DNS_TYPE_MX && answer_rr_dns.type == DNS_TYPE_MX)
{
    new_addr_info = tfDnsAllocAddrInfo();
    if (new_addr_info)
    {
        if (answer_rr_dns.rdlen >= 2)
        {
            // ... further parsing, including linking the new addrinfo to the list
        }
        else
        {
            // ... exit with error code
        }
    }
}
Heap Shaping – Done!

• The two allocation primitives are used to shape the heap

• Reliable overflow of tsDnsCacheEntry

• What can we overwrite with the CNAME pointer write primitive?
Pointer Write Limitations

• CNAME pointer written to address in tsDnsCacheEntry

  • Overflow is only alpha-numeric, with trailing null-byte
    • Can be used as segment MSB
  • Nothing placed in a strictly alpha-numeric address
  • Combine two alpha-numeric bytes => Non-alpha-numeric segment

\[
0x004B \ll 8 = 0x4B00
\]

\[
\begin{align*}
\text{Segment} & \quad 0x4141 \quad \text{Offset} \\
0x8C41 & \quad 008C:0041
\end{align*}
\]

• This allows us to overwrite heap utility functions
Overwriting a Far Call

• Far calls in x86 are encoded with a pointer
• Patching a far call using our primitive results in the CNAME buffer being executed

• We patch a far call in free() error flow
  • Called when metadata corruption is detected
Recap

- MX Name Buffer
- tsDnsCacheEntry
- dnscAddrInfoPtr
Recap

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>RDLENGTH</th>
<th>RDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>example</td>
<td>CNAME</td>
<td>14</td>
<td>004B:4141</td>
</tr>
</tbody>
</table>

```
mallocc(14);
```

```
1234:5678 => "EVIL.PAYLOAD"
```

```
sub_free:
... 1234:5678
    call cafe:d00d
```

```
addrinfo *dnscAddrInfoPtr
```

```
MX Name Buffer
```

```
AAAAAAAAAAAAAAAA
```

```
AAAAAAA
```

```
AAAAAA
```

```
AAAA
```

```
AAA
```

```
AA
```

```
A
```

```
E
```

```
V
```

```
I
```

```
L
```

```
7
```

```
P
```

```
A
```

```
Y
```

```
L
```

```
O
```

```
A
```

```
D
```

```
0
```

```
sub_free:
... 1234:5678
    call cafe:d00d
```

```
addrinfo *dnscAddrInfoPtr
```

```
MX Name Buffer
```

```
AAAAAAAAAAAAAAAA
```

```
AAAAAAA
```

```
AAAAAA
```

```
AAAA
```

```
AA
```

```
E
```

```
V
```

```
I
```

```
L
```

```
7
```

```
P
```

```
A
```

```
Y
```

```
L
```

```
O
```

```
A
```

```
D
```

```
0
```
Payload Trigger

- free() error flow will be triggered on overflown MX name free
- CNAME buffer contains crafted alpha-numeric shellcode
  - 2-stage decoder
Payload Trigger

• free() error flow will be triggered on overflown MX name free
• CNAME buffer contains crafted alpha-numeric shellcode
  • 2-stage decoder

• We have achieved arbitrary payload execution!
DEMO
Thanks for listening!

info@jsof-tech.com