CVE-2020-11901

JSOF Research Lab

Moshe Kol

Ariel Schön

Shlomi Oberman

JSOF

AUGUST 2020

Abstract

The JSOF research lab discovered a series of zero-day vulnerabilities, collectively known as Ripple20, in a widely used embedded TCP/IP stack called Treck.

In this paper we go in to the details of CVE-2020-11901, one of the most interesting vulnerabilities in Ripple20.

Behind the scenes, CVE-2020-11901 is comprised of 4 different vulnerabilities located in proximity to each other in the code and bundled in to one CVE. While every version of Treck using the vulnerable component is affected by a critical remote code execution vulnerability of these 4 (at the time of disclosure), some of the vulnerabilities affect all versions of Treck, and others affect only newer versions or only older versions.

Despite the fact that some of these vulnerabilities were fixed in the Treck stack over the years, all of the vulnerabilities existed “in-the-wild” at the time of disclosure, due to some vendors using older versions of the Treck TCP/IP stack and dropping ongoing support from Treck, and presumably also because these vulnerabilities have not been publicly recognized as security issues prior to the JSOF disclosure. This is just another artifact of the complexities and intricacies of vulnerabilities in a complex supply chain.

The most severe of CVE-2020-11901 are critical DNS client-side vulnerabilities located in Treck’s DNS resolver component, which received a CVSSv3.1 score of 9.1. If successfully exploited, they allow for a pre-authentication arbitrary remote code execution in the context of the network stack, often the OS kernel. These vulnerabilities are of particular interest because a sophisticated (e.g. nation state) attacker can potentially reply to a DNS request from outside of the corporate network, thus breaking network segmentation.

In addition to full details about the vulnerabilities themselves, we will also discuss details of exploitation on a Schneider Electric UPS Device.

JSOF

Contents

1 CVE-2020-11901 Overview **2**

2 DNS Protocol Primer **3**
2.1 The Basics . . . 3
2.2 DNS Packet Format . . . 4
2.3 Domain Names Format and Message Compression . . . 5

3 The Vulnerabilities **7**
3.1 Treck's DNS Resolver Internals . . . 7
3.2 Parsing DNS Responses . . . 8
3.3 DNS Label Length Calculation . . . 9
3.4 Vulnerability #1: Bad `RDLENGTH` Leads to Heap Overflow . . . 10
3.5 Vulnerability #2: From Integer Overflow to Heap Overflow . . . 10
3.6 Vulnerability #3: Read Out-of-Bounds . . . 13
3.7 Vulnerability #4: Predictable Transaction ID . . . 14
3.8 Closing Remarks . . . 14

4 Exploiting CVE-2020-11901 on a Schneider Electric UPS Device **15**
4.1 Exploit Strategy . . . 15
4.2 Treck Sheap (Simple Heap) Internals . . . 16
4.2.1 malloc() . . . 16
4.2.2 free() . . . 16
4.2.3 Summary . . . 17
4.3 Exploitation Technique . . . 17
4.3.1 Target Data Structure . . . 17
4.4 Heap Shaping . . . 18
4.4.1 Target Shape . . . 18
4.4.2 Temporary Allocation Primitive . . . 18
4.4.3 Persistent Allocation Primitive (or: Poking Holes) . . . 19
4.5 Overwriting a Far Call Destination . . . 20
4.6 Executing Shellcode . . . 20
4.6.1 Triggering `tsDnsCacheEntry` Free . . . 21
4.6.2 Executing Arbitrary Shellcode . . . 21
4.7 Putting It All Together . . . 21

5 References **23**

Chapter 1

CVE-2020-11901 Overview

CVE-2020-11901 is a single name for several critical client-side vulnerabilities in the DNS resolver of the Treck TCP/IP stack. If successfully exploited, they allow remote code execution for an unauthenticated attacker that is able to respond to a DNS query generated from an affected device. The vulnerabilities can potentially be exploited across network boundaries, making them extremely dangerous.

The vulnerabilities mostly stem from an incorrect DNS label length calculation. One vulnerability is triggered by specifying small `RDLENGTH` value, and another leverages DNS message compression scheme in order to achieve an integer overflow.

In this paper, we detail the vulnerabilities and we will explain an exploit on a Schneider Electric UPS device.

Chapter 2

DNS Protocol Primer

As the vulnerabilities reside in the DNS resolver component of the network stack, we'll take a quick look into the DNS protocol.

2.1 The Basics

The domain name system (DNS) is a networking infrastructure protocol that enables TCP/IP applications to map *domain names* into IP addresses. The domain name space is hierarchical: it starts from an unnamed root, branching into top-level domains (TLDs) such as, `com`, `org`, `edu`, and continues to branch from there. A *fully-qualified domain name* (FQDN) represents a path from a root to a node in the DNS tree, and it is written from right-to-left, starting with the node and ending with the root. For instance, `www.example.com.` (note the trailing dot), represents this path:

.(root) ⟶ `com` ⟶ `example` ⟶ `www`

A TCP/IP application converts a name into an address using a DNS *resolver*. The resolver issues a query to one or more configured DNS servers, requesting information about the domain in question. Once a DNS response arrives, it is parsed and the information is passed to the application, and often cached for additional use. Treck TCP/IP has a set of public API functions for interaction with the DNS resolver, as we will see later.

Information about domains is distributed among *name servers*. Every name server manages one or more *zones*. A zone is a sub-tree of the hierarchical domain name space that is administrated separately. A *zone file* stores information about the zone in *records*. Each record has a name, value, type, class and TTL. The interpretation of the record's value depends on its type and class.

In practice, the only used class is the internet class `IN` (1). Common record types within this class include:

Type	Description
A	IPv4 address.
AAAA	IPv6 address.
MX	Mail exchange record. The Record's value is a tuple of preference value and a domain name of a host capable of accepting emails.
CNAME	Used to define *aliases*.
NS	Specifies an authoritative name server for the domain. The record's Value is a domain name.

Header (12 bytes)
Questions (var)
Answers (var)
Authority (var)
Additional (var)

Figure 2.1: DNS high level structure

Transaction ID								+0
QR	OPCODE	AA	TC	RD	RA	Z	RCODE	+2
QDCOUNT								+4
ANCOUNT								+6
NSCOUNT								+8
ARCOUNT								+10
								+12

Figure 2.2: DNS header format

MX Records Because some of the code presented later in the paper concerns MX records, we will elaborate more about them. MX (Mail eXchange) records are an essential part of the way emails work. When sending an email using the familiar `user@host` notation, the mail client issues a DNS query of type MX to resolve `host`. As an answer, the DNS server returns a domain name of a mail server (also called "MX hostname"). Each MX hostname is associated with a 16-bit integer called *preference value* used to prioritize hostnames. A smaller preference value has a higher priority.

Since a domain name of the mail server is returned (rather than an IP address), the client's resolver needs to further resolve the MX hostname with a second request in an order to get an IP address. To reduce latency, most DNS servers hand-in the IP address along with name. Nonetheless, this "double-query" functionality should be supported by the resolver.

2.2 DNS Packet Format

DNS messages are generally transmitted over UDP [1]. The high level structure of a DNS message is shown in figure 2.1. A DNS message (query or response) has has a fixed-length header of 12 bytes, followed by 4 variable-length sections.

The DNS header format is shown in figure 2.2. The transaction ID is a 16-bit field set by the resolver (client) to match the response with the query. The `QR` field is a bit specifying whether the packet is a query (`QR=0`) or response (`QR=1`). `RCODE` holds the response code, where a non-zero value indicates an erroneous conditions. The `QDCOUNT`, `ANCOUNT`, `NSCOUNT` and `ARCOUNT` specify the number of entries in each of the 4 sections following the header, respectively.

[1] There is an option to transfer DNS over TCP but we will not discuss it here

NAME (var)	TYPE (2 bytes)	CLASS (2 bytes)

Figure 2.3: Question section format

NAME (var)	TYPE (2 bytes)	CLASS (2 bytes)	TTL (4 bytes)	RDLENGTH (2 bytes)	RDATA (var)

Figure 2.4: DNS resource record format

Each entry of the questions section, referred to as "question", has a specific format, shown in figure 2.3. The *query name* field specifies the name being looked up. It follows a specific format we will describe later. The client specifies the type of the record of interest in the *query type* field.

The entries of answers, authority and additional sections are called *resource records*. These records will be major player in our vulnerabilities. The format is depicted in figure 2.4. Each resource-record is associated with a `NAME`, to which the record refers, as well as with a `TYPE` and a `CLASS`. The `TTL` (time-to-live) field specifies the number of seconds the resource record is allowed to be cached by the client. The `RDATA` field contains the record's value. Its length is specified in the `RDLENGTH` field.

2.3 Domain Names Format and Message Compression

Last but not least, the binary format of a *name*. The format is specified in [2, sections 3.1, 4.1.4], and it is relatively simple. A domain name is a sequence of *labels*, terminated by the empty label. Each label is preceded by a length byte. The maximum label length is 63 bytes, and the empty label has length 0. As an example, we will encode the domain name `www.example.com`:

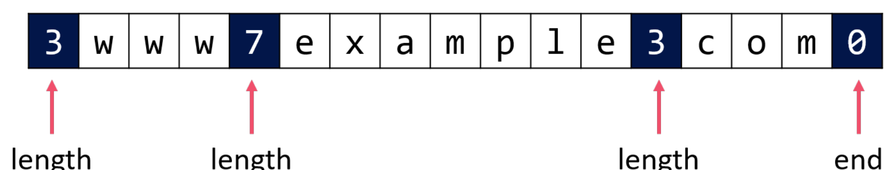

Figure 2.5: Domain name encoding

The conversion from the binary format to the ASCII format is straight-forward: replace all length bytes with dots (.) except the first and the last.

A typical DNS response includes the same domain name or a part of it several times. Consider a query of type A for the domain name `www.example.com`. Some DNS servers include `www.example.com` two times in their response:

```
0000   e3 70 81 a0 00 01 00 01 00 00 00 00 03 77 77 77   .p..........www
0010   07 65 78 61 6d 70 6c 65 03 63 6f 6d 00 00 01 00   .example.com....
0020   01 03 77 77 77 07 65 78 61 6d 70 6c 65 03 63 6f   ..www.example.co
0030   6d 00 00 01 00 01 00 01 0d f6 00 04 5d b8 d8 22   m...........].."
```

Figure 2.6: DNS response packet in which `www.example.com` is repeated twice

The first `www.example.com` is included as part of the `QUESTIONS` section, while the second is included as part of the `ANSWERS` section. Much more repetition is possible with queries for

domains that have multiple `A` or `CNAME` records (you can try resolving `www.youtube.com` by yourselves).

To reduce the size of DNS messages, a compression scheme can be used. In the scheme taken by the designers of DNS, compression is achieved by replacing a sequence of labels with a pointer to a previous occurrence of the same sequence. The pointer is encoded in two bytes, the first of them begins with two high bits `11`, and the other 14 bits specify an offset from the start of the DNS header.

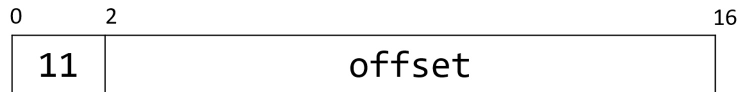

Figure 2.7: Compression pointer format

To illustrate the concept, suppose that at offset `0x10` from the start of a DNS response packet there is a sequence of labels corresponding to `example.com`:

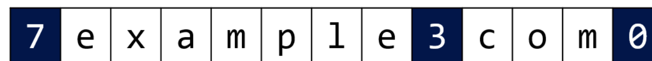

Figure 2.8: Domain name encoding of `example.com`

In order to encode, for example, the domain name `x.y.z.example.com`, we can encode it directly or utilize the compression scheme. In the latter case, the encoded domain name would look like this:

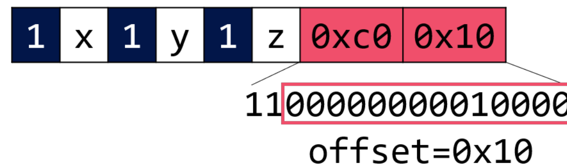

Figure 2.9: Encoding of `x.y.z.example.com` using compression

This instructs the DNS server or client to concatenate the sequence of labels beginning at offset `0x10` to the sequence `x.y.z.`. The expanded domain name is `x.y.z.example.com`, as expected.

Returning to the example of resolving `www.example.com`, let's look at a real world case. This is part of a response from Google's `8.8.8.8` DNS server, which uses compression:

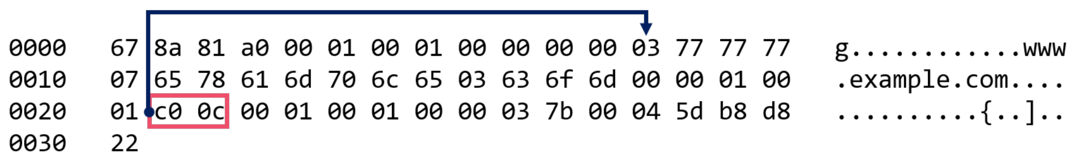

Figure 2.10: DNS response packet in which compression is used for the second occurrence of `www.example.com`

As you can see, the name of answer resource record is replaced by a pointer to offset `0xc`, which is the start of the `www.example.com` domain name.

Chapter 3

The Vulnerabilities

In this section we will explain the vulnerabilities known as CVE-2020-11901.

3.1 Treck's DNS Resolver Internals

A TCP/IP application running on top of the Treck TCP/IP stack interacts with the DNS resolver using high-level APIs such as `tfDnsGetHostByName` and `tfDnsGetMailHost`. When a request is made using these functions, a `tsDnsCacheEntry` structure is created in order to record some information about it (a new cache entry is not created if a cache-entry describing the request is already present). Some of `tsDnsCacheEntry` fields are shown below:

```
typedef struct tsDnsCacheEntry {
    struct tsDnsCacheEntry TM_FAR * dnscNextEntryPtr;
    struct tsDnsCacheEntry TM_FAR * dnscPrevEntryPtr;
    // Pointer to chain of address information structures.
    struct addrinfo        TM_FAR * dnscAddrInfoPtr;
    // ...
    // String used in DNS query packet (in DNS label format)
    ttCharPtr                       dnscRequestStr;
    // Error code (if any) returned from socket of DNS server
    int                             dnscErrorCode;
    // ...
    // Indicates if this entry is completed and has been retrieved by the user
    tt16Bit                         dnscFlags;
    // ...
    // ID of the last request sent that was associated with this cache entry
    tt16Bit                         dnscRequestId;
    // Query type (Name, MX, reverse)
    tt16Bit                         dnscQueryType;
};
```

`tsDnsCacheEntry` structures are stored in a doubly-linked list. They hold information about the DNS request being made, such as the request string (`dnscRequestStr`), the transaction ID (`dnscRequestId`) and the query type of the request (A, AAAA, MX, ...) with `dnscQueryType`. This struct also holds information about the response, most notably using a chain of address information structures (pointed to by `dnscAddrInfoPtr`).

An `addrinfo` is a small structure containing the following fields:

```
struct addrinfo {
    int                     ai_flags; // AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST
    int                     ai_family; // PF_INET, PF_INET6
    int                     ai_socktype; // SOCK_xxx (not used)
    int                     ai_protocol; // 0 or IPPROTO_xxx (not used)
    unsigned int            ai_addrlen; // length of ai_addr
```

```
7  #define ai_mxpref ai_addrlen
8      char TM_FAR *           ai_canonname; // canonical name for nodename
9  #define ai_mxhostname ai_canonname
10     struct sockaddr TM_FAR * ai_addr; // binary address
11     struct addrinfo TM_FAR * ai_next; // next structure in linked list
12 };
```

An addrinfo structure holds information about addresses returned by the DNS server. The exact information it holds depends on the query type. When the query type is A or AAAA (requesting IP address for a name), this structure holds IP address information (ai_addr) along with a canonical name (ai_canonname), if supplied at the response. If the query type is MX (mail exchange), the MX preference value is stored in place of ai_addrlen and the MX hostname is store in place of ai_canonname.

3.2 Parsing DNS Responses

Treck's DNS resolver processes incoming DNS responses in a function called tfDnsCallback. This function accepts tsDnsCacheEntry structure as argument and is responsible for parsing a DNS response message and caching information about the queried name.

During the parsing of the ANSWERS section from the DNS response packet, the code allocates an addrinfo and stores relevant information in it. Let us take a closer look at the piece of (pseudo-)code parsing MX records:

```
1  if (RDLENGTH <= remaining_size) {
2      /* compute the next resource record pointer based on the RDLENGTH */
3      labelEndPtr = resourceRecordAfterNamePtr + 10 + RDLENGTH;
4      /* type: MX */
5      if (cacheEntryQueryType == DNS_TYPE_MX && rrtype == DNS_TYPE_MX) {
6          addr_info = tfDnsAllocAddrInfo();
7          if (addr_info != NULL && RDLENGTH >= 2) {
8              /* copy preference value of MX record */
9              memcpy(&addr_info->ai_mxpref,resourceRecordAfterNamePtr + 10, 2);
10             /* compute the length of the MX hostname */
11             labelLength = tfDnsExpLabelLength(resourceRecordAfterNamePtr + 0xc, dnsHeaderPtr,
               ↪  labelEndPtr);
12             addr_info->ai_mxhostname = NULL;
13             if (labelLength != 0) {
14                 /* allocate buffer for the expanded name */
15                 asciiPtr = tfGetRawBuffer((uint)labelLength);
16                 addr_info->ai_mxhostname = asciiPtr;
17                 if (asciiPtr != NULL) {
18                     /* copy MX hostname to `asciiPtr` as ASCII */
19                     tfDnsLabelToAscii(resourceRecordAfterNamePtr + 0xc, asciiPtr,
                       ↪  dnsHeaderPtr, 1, 0);
20                     /* ... */
21                 }
22                 /* ... */
23             }
24             /* ... */
25         }
26         /* ... */
27     }
28 }
```

As we can see, the resolver performs the following steps to handle incoming MX answers:

1. Allocate a new addrinfo struct.
2. Store the MX preference value in the ai_mxpref field.

3. Compute the length of the MX hostname using `tfDnsExpLabelLength`.
4. Allocate a new buffer based on the returned length `labelLength`.
5. Convert the raw MX hostname to ASCII using `tfDnsLabelToAscii`, and store the result into the newly allocated buffer `asciiPtr`.

Notice that `tfDnsLabelToAscii` is **not** aware of the size of the allocated buffer `asciiPtr`. The `tfDnsLabelToAscii` function will copy only alphanumeric and '-' characters to the destination buffer, until non-alphanumeric or non-hyphen character is reached. Furthermore, if the 4th and 5th arguments passed to it are 1, 0 respectively, `tfDnsLabelToAscii` function will honor the DNS compression scheme. We can see from the code that if the length returned by `tfDnsExpLabelLength` is too small, `tfDnsLabelToAscii` will overflow the allocated buffer, with an alphanumeric overwrite only.

At this stage we will investigate the length calculation function `tfDnsExpLabelLength` in depth.

3.3 DNS Label Length Calculation

Below is a pseudo-code for `tfDnsExpLabelLength`:

```
1  tt16Bit tfDnsExpLabelLength(tt8BitPtr labelPtr, tt8BitPtr pktDataPtr, tt8BitPtr labelEndPtr){
2      tt8Bit currLabelLength;
3      tt16Bit i = 0, totalLength = 0;
4      tt8BitPtr newLabelPtr;
5
6      while (&labelPtr[i] < labelEndPtr && labelPtr[i] != 0) {
7          currLabelLength = labelPtr[i];
8          if ((currLabelLength & 0xc0) == 0) {
9              totalLength += currLabelLength + 1;
10             i += currLabelLength + 1;
11         } else {
12             if (&labelPtr[i+1] < labelEndPtr) {
13                 newLabelPtr = pktDataPtr + (((currLabelLength & 0x3f) << 8) | labelPtr[i+1]);
14                 if (newLabelPtr < labelPtr) {
15                     labelPtr = newLabelPtr;
16                     i = 0;
17                     continue;
18                 }
19             }
20             return 0;
21         }
22     }
23     return totalLength;
24 }
```

`tfDnsExpLabelLength` calculates the length of the DNS label pointed to by `labelPtr`. It's basic operation is to sum-up all length bytes of the given label (domain name), while honoring compression pointers. It accepts a pointer to the start of the packet data buffer in `pktDataPtr` and an end-pointer in `labelEndPtr`. Using the end-pointer, the function ensures that out-of-bound data is not read. The pointer to the start of the packet data buffer, `pktDataPtr`, is necessary to support the DNS compression scheme described earlier.

The function maintains the total label length in the `totalLength` variable. It continues to read the current label length (`currLabelLength`) as long as it is greater than 0. If the current label length ≤ 63, it is added to the `totalLength` variable. Otherwise, it means that a compression pointer follows (one of the higher two bits is set). The function reads the compression pointer and computes the new label pointer in `newLabelPtr`. Then, it verifies

that the new label pointer points *before* the current label pointer. If this is the case, the length of the new label pointed to by the compression pointer, is computed in the same way, and added to `totalLength`.

There are several issues with this implementation:

1. The type of the `totalLength` variable is `tt16Bit` (typedef for `unsigned short`). If possible, overflowing the `totalLength` variable would result in a small number returned by `tfDnsExpLabelLength`. A buffer would then be allocated based on that number, and the buffer could be overflown by `tfDnsLabelToAscii`.

2. It does not honor the 255 maximum domain name length as specified in [2].

3. It does not validate the characters used in the domain name itself: they should be alphanumeric and '-' only.

4. It treats label lengths with most-significant-bits `01` and `10` as if they specify compression pointer. According to [2], only two high bits `11` should specify compression, and the other cases are reserved for future use. This behavior could come handy in some cases, as `0x4130` for example is a valid compression pointer and also contains valid alpha-numeric characters.

In addition, notice that the function stops processing when `&labelPtr[i]` reaches the end-pointer `labelEndPtr`. It does not return an error in this case, but simply the current `totalLength` value.

3.4 Vulnerability #1: Bad RDLENGTH Leads to Heap Overflow

Recall that in the section 3.2 we discussed the handling of MX resource records. An end-pointer is passed to the `tfDnsExpLabelLength` and is calculated based on the `RDLENGTH` field:

```
labelEndPtr = resourceRecordAfterNamePtr + 10 + RDLENGTH;
```

The `RDLENGTH` is a 16-bit field part of the resource record (recall section 2.2). This field specifies the number of bytes of the `RDATA` field, and it is attacker-controlled. An attacker can specify a small `RDLENGTH` value, causing the length calculation to stop prematurely. This results in `tfDnsExpLabelLength` returning small length number. As explained earlier, this leads to heap-based buffer overflow vulnerability. This vulnerability only exists in newer versions of the Treck TCP/IP stack, and affects the latest version at the time of disclosure. We do not know the exact version when this vulnerability was introduced.

3.5 Vulnerability #2: From Integer Overflow to Heap Overflow

Treck's resolver accepts DNS response packets over UDP with a size of up to ($\leq$) `0x5b4` (1460). As explained earlier, the total label length maintained by `tfDnsExpLabelLength` is of type `unsigned short` (16-bit width). So we can now ask: is it possible to encode a domain name within a single DNS response packet so that the domain name length is expanded (as calculated by `tfDnsExpLabelLength`) to 65536 (2^{16}) or more?

To achieve this kind of length amplification over UDP[1], we certainly must use the compression feature of DNS. Keep in mind that with a compression pointer we can only jump *backwards* from our current label pointer (see section 3.3), making the task more challenging.

[1] At least in the versions of Treck at our disposal, DNS over TCP is not supported. Had it been supported, our task would have been much easier because DNS over TCP allows large packet sizes of up to 65536 bytes.

Since `tfDnsExpLabelLength` does not enforce the validity of characters in a label, we can overload the bytes of a label with useful information. "Overload" in this context means that bytes normally part of the label contents have an extra role in which they are interpreted as length bytes or compression pointers under different circumstances.

The Basic Technique The following example introduces the basic technique we used in our proof-of-concept. Consider the encoded name shown in figure 3.1.

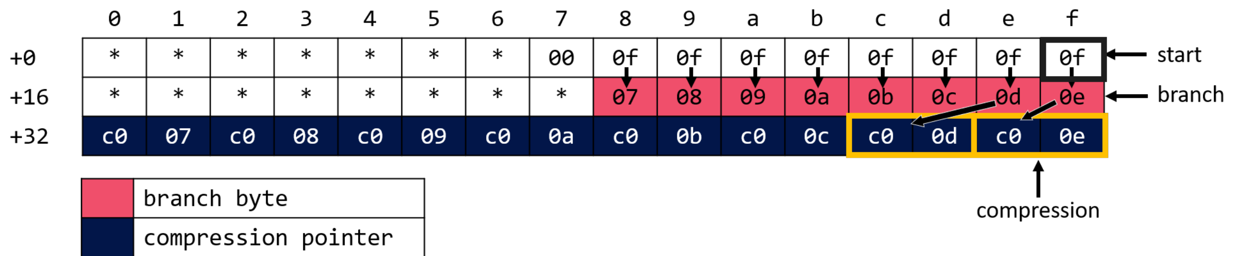

Figure 3.1: Tiny example of using compression pointers. Cells with * can be filled with any byte value

The technique is easier to understand if we think about the encoded name as a matrix. In this tiny example, each row has size of 16 bytes. This number was chosen so that it will be easier to follow.

Suppose that the calculation of the name's length starts at offset `0xf`, the last byte of the first row. This can be achieved by placing the name somewhere in the DNS response and then refer back to it using a compression pointer (more about this below). With this assumption, `tfDnsExpLabelLength` would read a label length byte of `0x0f`, and advance the input head to the last byte of the second row (staying on the same column). Then it reads a label length of `0x0e` and advance the input head to the second-to-last byte of third row (offset `0xe` of the third row). At this point it reads a compression pointer. The pointer offset is `0xe` which is strictly less than our starting point (at offset `0xf`), so the process repeats from offset `0xe`. The next compression read is to offset `0xd`, and so on. The length calculation stops when it reaches offset `0x07`, because we placed a null-byte there in this example.

Of course we can add more rows between the starting point and the part of the branch bytes. In figure 3.2 we demonstrate the addition of 5 extra rows.

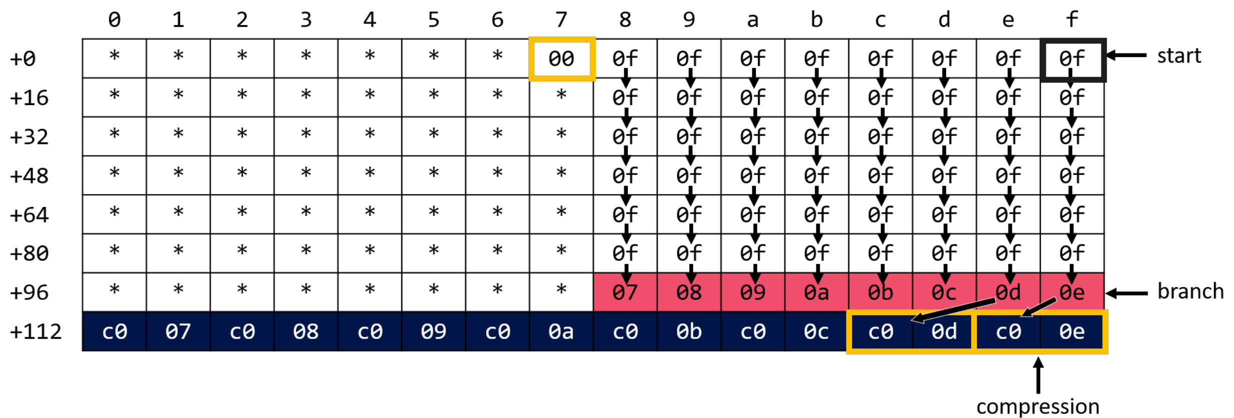

Figure 3.2: Tiny example of using compression pointers with 5 extra rows

The extra rows we added have no particular role except extending the name as much as possible. To make the explanation easy for the rest of the section, we shall introduce some definitions. A **lane** is a column in this matrix, terminated by a **branch byte**. A branch byte routes the lane to the appropriate compression pointer - paving the way for the next lane.

In the previous example we used 8 lanes. We can improve this situation by including more compression pointers at the second-to-last row and more branches at the third-to-last row. Building on that idea, we reach the following:

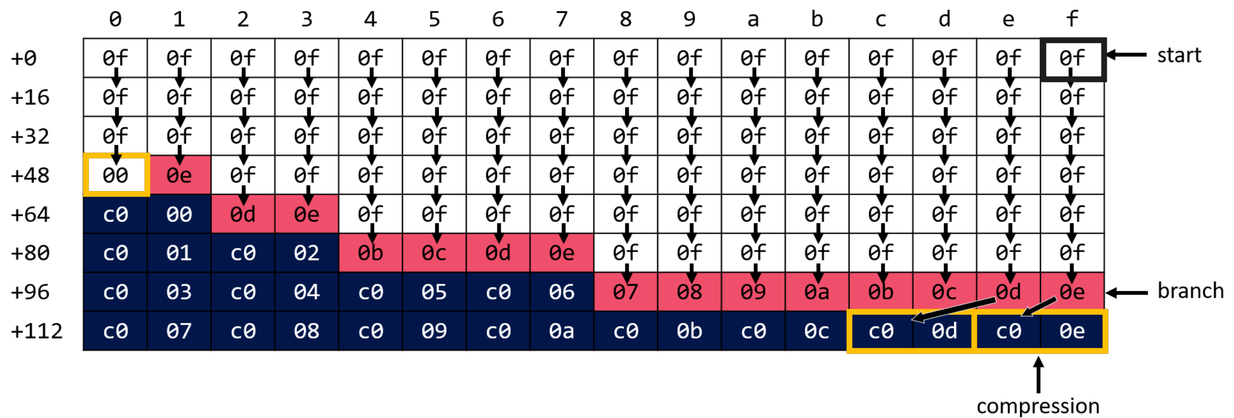

Figure 3.3: Full illustration of the basic technique

Now we use all of the 16 lanes at our disposal. At the last row we have 8 compression pointers and so 8 branch bytes at the second-to-last row. This leaves us with another 4 compression pointers to be used at the second-to-last row. These compression pointers are reached by the branch bytes at the third-to-last row (offsets 4-7), which leaves us with a place for another 2 compression pointers, and so on. The last lane (which begins in the first row offset 0), is terminated with a null byte (at offset 0 of the forth row).

By using this construction, in this example, we reached a total length of 1502. This is pretty neat if you consider the fact that the name itself only occupies 128 bytes.

Proof-of-Concept In the simple example presented above we did not reach a state where the `totalLength` variable is overflowed. However, we are not far from it. We have two tools we can use to maximize the total length:

1. Maximize the size of a row in the matrix. Because the maximum length byte allowed is 63 (=`0x3f`), we can use rows which are up to 64 bytes in length.

2. Pad lanes with more extra rows.

We used these two tools to maximize the total length. With 64 lanes, this construction overflows the 16-bit `totalLength` variable in `tfDnsExpLabelLength`.

We are almost done. Until now, we hid some information under the rug:

1. Where this "long name" is placed in the DNS packet.

2. How we begin the length calculation from a specific spot in this long name.

3. How we overflow, if `tfDnsLabelToAscii` only copies alphanumeric characters in to the buffer, but our matrix is comprised of non-alphanumeric characters.

The answer to all of these is in how we constructed the packet. A DNS packet can have multiple questions, so we added another question, the name of which is our matrix construction. The MX hostname then consists of a label consisting of a chosen alpha-numeric payload followed by a single compression pointer that will land us exactly on the required starting byte.

The result is heap-based buffer overflow vulnerability: due to the integer overflow, `tfDnsExpLabelLength` returns a `labelLength` of small size. Based on this size, a buffer is allocated on the heap. `tfDnsLabelToAscii` is then asked to copy the encoded name as ASCII, overflowing the buffer just-allocated with the payload at the beginning of the MX hostname.

This vulnerability affects the latest Treck version at the time of disclosure (version 6.0.1.6).

3.6 Vulnerability #3: Read Out-of-Bounds

In older versions of Treck, there was a read OOB vulnerability during name parsing. The following snippet of pseudo-code shows the MX parsing logic as it was in older versions:

```
1  if (cacheEntryQueryType == DNS_TYPE_MX && rrtype == DNS_TYPE_MX) {
2      addr_info = tfDnsAllocAddrInfo();
3      if (addr_info != NULL) {
4          /* copy preference value of MX record */
5          memcpy(&addr_info->ai_mxpref, resourceRecordAfterNamePtr + 10, 2);
6          /* compute the length of the MX hostname */
7          labelLength = tfDnsExpLabelLength(resourceRecordAfterNamePtr + 0xc, pktDataPtr);
8          addr_info->ai_mxhostname = NULL;
9          if (labelLength != 0) {
10             /* allocate buffer for the expanded name */
11             asciiPtr = tfGetRawBuffer(labelLength);
12             addr_info->ai_mxhostname = asciiPtr;
13             if (asciiPtr != NULL) {
14                 /* copy MX hostname to `asciiPtr` as ASCII */
15                 tfDnsLabelToAscii(resourceRecordAfterNamePtr + 0xc, asciiPtr, pktDataPtr);
16                 /* ... */
17             }
18 /* ... */
```

In this older version, we can see that there are no checks on the packet buffer. In particular, there's no end-pointer being calculated and passed to `tfDnsExpLabelLength`. In this version, `tfDnsExpLabelLength` have the exact same operation as explained in section 3.3, except that it doesn't have bounds check. Instead, it will iterate over the length bytes until a null-byte is reached, possibly crossing buffer bounds.

This issue could result in a denial-of-service vulnerability, if, for instance, the function reads from an unmapped page while iterating over the length bytes.

More interestingly, the issue could result in an information leakage vulnerability: `tfDnsLabelToAscii` has no bounds check either, so it could copy OOB bytes into the destination buffer. This means that when parsing MX responses, data from the heap could be interpreted as an MX hostname. The MX hostname just read is later resolved by the client in an attempt to get an IP address (see paragraph about MX records in 2.1). The result is that it is potentially possible to leak data from the heap as part of the MX hostname being resolved.

The issue affects Treck versions at least 4.7, and was fixed in later versions[2]. We don't know the exact version in which Treck fixed the issue. Suffice to say that the vulnerability still impacts devices in-the-wild: Some vendors stop updating the Treck stack, yet they continue to ship their products to other vendors in the supply chain. The unfortunate result is that there are embedded devices that contain an older version of Treck, and are thus vulnerable to this issue.

[2] the fix for the read OOB was a bad fix and caused the bad `RDLENGTH` vulnerability, described in section 3.4.

3.7 Vulnerability #4: Predictable Transaction ID

Another vulnerability, only seen in earlier versions of the network stack, is that the DNS transaction ID is incremented serially, and starts at 0, making it easily guessable. This means that the attacker might not need to execute complex man-in-the-middle attack in order to find the value of the transaction ID. This flaw makes our DNS exploits easier and can lead to other consequences. Despite being fixed over the years in the Treck TCP/IP stack it still exists in devices in the wild, in cases where companies stopped support for the Treck stack or other cases.

3.8 Closing Remarks

In this chapter, we have demonstrated a heap-based buffer overflow in the DNS resolver code using two different techniques: bad `RDLENGTH` and integer overflow. The function responsible for the copy (`tfDnsLabelToAscii`) will only copy alphanumeric and '-' (hyphen) characters to the destination buffer, until non-alphanum or non-hypen character is reached. This means that we have alphanumeric overwrite, and that we can stop overflowing at our will using an invalid character. In the next chapter we will see how we leveraged this primitive into a remote code execution.

The two heap overflow vulnerabilities (and the read OOB) were demonstrated in the parsing logic of the MX type. However, the same vulnerabilities are present also during the processing of CNAME resource records and PTR resource records. Considering the fact that CNAME records are supported by the DNS resolver no matter what the original query type was, this means we can cause heap-based buffer overflow with every type of DNS query generated by an affected device.

The two heap overflow vulnerabilities affect the latest Treck version at the time of disclosure (version 6.0.1.66). The read out-of-bounds and predictable transaction ID vulnerabilities were fixed in later versions of the network stack, we don't know the exact version.

Chapter 4

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

In this part we will show how we exploited CVE-2020-11901 on a Schneider Electric UPS device and achieved remote code execution. The specific model we used is an APC Smart-UPS 750 (SMT750I/ID18/230V) running firmware v09.3 together with a Network Management Card (NMC) version 2 (model AP9630) running AOS version v6.8.2 (latest at the time of disclosure).

A UPS (Uninterruptible Power Supply) device is designed for use in enterprise networks, data centers, and mission-critical systems. It ensures, using an embedded battery, that devices connected to it will not suffer from power outages or fluctuations. As such, exploiting a UPS device remotely, can have disastrous consequences.

Many of the UPS devices manufactured by Schneider Electric can be connected to an IP network via a network card called NMC. The NMC runs on an x86-based processor and it runs the Treck TCP/IP as its network stack. However, this processor implements segmentation in a unique manner: instead of calculating a linear address by shifting the segment left by 4-bits like most x86-based architectures, it shifts the segment left by 8-bits. The operating system, network stack, and APC application all run in 16-bit real mode.

Having only a single device at the beginning of our research, and worried about ruining the device, we decided to attempt research and exploit development on the device without JTAG or other debug capabilities. Our main sources of information were static analysis, and crash dumps containing a small stack trace and some register values.

Our proof-of-concept exploit uses the bad `RDLENGTH` vulnerability to run a shellcode which shuts down all UPS outlets, turning off any device relying on it for its operation. We will now describe our exploit - the primitives established and the techniques used. The exploit should be easily adaptable to devices using a vulnerable version of the Treck stack with the same heap configuration.

4.1 Exploit Strategy

When it comes to exploiting heap overflow vulnerabilities, there are generally two routes:

- Overwrite heap meta-data information.
- Overwrite application-specific data structures.

The former is usually considered more generic, as it does not require overflowing any specific data structures, only memory blocks, possibly of a specific size.

In the previous white paper we detailed exploitation of CVE-2020-11896 [3], where we chose to exploit the device by corrupting heap metadata. We will now examine the Treck heap

implementation present on the UPS device to determine if this is a suitable plan in this case as well.

4.2 Treck Sheap (Simple Heap) Internals

Treck uses a proprietary, highly configurable heap implementation. In this specific configuration, the heap is different from the heap described in [3].

During device boot, the heap is created statically in a hard-coded address, initially comprised of 4 free blocks, each `0xAF00` bytes in size. The heap features two bins, each comprised of a free-list: one for "small" blocks under `0x400` bytes, and one for larger blocks. Free-list blocks are saved in a singly linked list, where each block contains a "next" pointer and its size in dwords. The size is saved both at the beginning (prefix) and the end of the block (postfix). Allocated blocks contain prefix and postfix size fields as well.

4.2.1 malloc()

The entry point for allocation in the Treck heap is `tm_kernel_single_malloc`. This is the low-level heap function invoked by more abstract allocation functions such as `tfGetRawBuffer` for data buffers and `tfGetShareBuffer` for packets. `tm_kernel_single_malloc` receives a single size argument. Internally, the function will call `tfSheapGetFreeListBlock` which will iterate the free-list bins. If the requested size fits the small free-list bin (`size <= 0x400`), it will try to allocate from that bin first, resorting to the big bin if no small block was found.

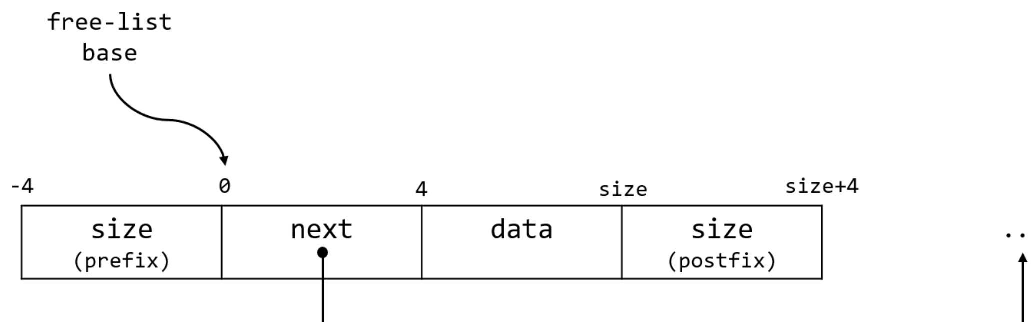

Figure 4.1: Free list illustration

The function `tfSheapGetFreeListBlock` is responsible for selecting a free block and will iterate over the entire free-list. For every block it will check if the prefix and postfix size fields match by adding the prefix size to its base pointer and reading the postfix size field from the resulting address. It will do this for every block in the list, regardless of whether the block is allocated and returned to the caller. If the sizes are not equal, an assert will fail and the machine will crash. Assuming everything went well, the tightest fitting block for the requested size will be returned. `tm_kernel_single_malloc` will then check if the returned block is bigger than the requested allocation size - if so, it will use only as much memory as it requires, making a new free block out of the remaining memory.

4.2.2 free()

The entry point for freeing memory in the Treck heap is `tm_kernel_free`. It accepts a pointer to an allocated buffer and does the following actions:

1. Make sure prefix and postfix sizes for the allocated block are valid.
2. Iterate both free-list bins:

(a) For each free block, make sure prefix and postfix sizes are valid.

(b) If neighboring free blocks are found, coalesce them into one free block.

3. Make sure the current freed size isn't larger than the total allocated data (global variable).

4. Add the new free block into the correct free-list bin.

4.2.3 Summary

Since the vulnerability allows us to overflow a heap buffer only with alpha-numerical data, it can prove hard to corrupt heap metadata in a way that will not cause a system crash[1]: The size field is 4 bytes long, and the smallest value[2] we can write is `0x30303030`, a large number, making it hard for us to fake a postfix size that will pass validation[3]. We can overwrite an allocated block without worrying about unexpected crashes only until that block itself is freed, at which point an error flow will be triggered including an assert failure and a device crash.

Because of the checks implemented in this heap configuration, we decided to exploit the device by overwriting application-specific data structures allocated on the heap rather than the heap metadata itself.

4.3 Exploitation Technique

The DNS name parsing vulnerabilities exist in all types of queries, either through the use of a CNAME record (which is always allowed) or though name fields in MX or PTR records. The attack surface we chose is the MX response parsing logic, for a few reasons.

First, when the device boots with a mail recipient configured, it will send out 3 MX requests by default before giving up. Because we chose to overflow application-specific data structures, we need to shape the heap so that we overflow our desired data structure. Shaping is more straight forward with multiple DNS request-response pairs. On top of that, since we don't have any debug interfaces, it is useful crash the device anyway to get it into a relatively deterministic device state. Therefore, the attack scenario chosen is to cause a crash by overflowing the heap with garbage upon any DNS request. After the device reboots in a clean state and starts sending MX requests, we will execute our exploit logic.

4.3.1 Target Data Structure

During DNS response parsing all sorts of data structures are allocated, including the previously mentioned `tsDnsCacheEntry` and `addrinfo` structures , among others. We chose `tsDnsCache-Entry` as our overflow target.

Especially interesting is the parsing of the CNAME answer record: the record parsing loop saves pointers to the CNAME record and returns to parse it after all other answer records were parsed. Unlike other names where a new `addrinfo` struct is allocated, CNAME records are parsed as follows:

```
// ...
unsigned short length;
addrinfo far * first_addr_info;
char far * cname_label_buffer = 0;

```

[1] According to the metadata validations specified earlier.

[2] We can also use bytes smaller than `0x30`, such as '.', but this doesn't solve the problem and imposes new limitations.

[3] We succeeded in doing this, but the limitations were complex and this did not seem like a promising exploitation plan

```
6  if (found_cname) {
7    // Get the first addr info from `tsDnsCacheEntry`
8    first_addr_info = t_dns_cache_entry.dnscAddrInfoPtr;
9    if (first_addr_info != NULL) {
10     // get CNAME name length from the packet (the usual vulnerabilities here...)
11     length = tfDnsExpLabelLength(cname_rdata_ptr, packet_ptr, cname_rdata_end_ptr);
12     if (length != 0) {
13       // allocate
14       cname_label_buffer = tfGetRawBuffer(length);
15       if (cname_label_buffer != NULL) {
16         // copy to new buffer
17         tfDnsLabelToAscii(cname_rdata_ptr, cname_label_buffer, packet_ptr, 1, 0);
18         first_addr_info.ai_canonname = cname_label_buffer;
19       }
20     }
21   }
22 }
```

CNAME record names don't get a new `addrinfo`, but are placed in the first one pointed to by `tsDnsCacheEntry`. Therefore, if we manage to overflow `tsDnsCacheEntry->dnscAddrInfoPtr`, we can cause writing of a dword to a location of our choosing (as long as the address is alphanumerical, like our overflow); The dword written will be a pointer to a newly allocated buffer containing our CNAME. This is a strong exploitation primitive, and thus the one we ended up using.

4.4 Heap Shaping

By shaping the heap using the two allocation primitives specified later, we managed to overflow the `tsDnsCacheEntry` struct without corrupting any free blocks. It is worth noting that in the case of this UPS device, failure to shape the heap will most likely cause the network card to crash, which has no noticeable effect on UPS operation. The network card reboots quickly and automatically, allowing several additional exploitation attempts [4].

In our PoC exploit, we initiated a TCP handshake through an open port to allow easier shaping. This is not necessary, as the first 2 MX packets can be crafted to do the entire shaping job, as shown below.

4.4.1 Target Shape

In order to overwrite this struct, the overflowing MX record needs to be allocated before the `tsDnsCacheEntry` struct. Chronologically, the cache entry is allocated on DNS request generation, before the MX record name buffer. Luckily, this heap is sophisticated enough to favor tightest-fit for allocations. This means we can create a pattern will result in allocations in the correct order for exploitation. A `tsDnsCacheEntry` allocation is `0xA0` bytes large, and we control our MX record size completely. The pattern that would fit our needs is shown in figure 4.4.1. The blocks labeled "allocated" are required to prevent free block coalescing and act as separators.

4.4.2 Temporary Allocation Primitive

The "best" allocation primitive in the DNS parsing code is name buffer allocation. For every answer (MX, PTR, CNAME) that has a DNS name in its `RDATA` field, `tfGetRawBuffer` is used to allocate a buffer for that name. It is important to note that along with this buffer, an `addrinfo` struct of size `0x38` bytes is also allocated. This primitive allows us to allocate a

[4] if the device is crashed too many times, eventually, the it will reboot in to a different state, resulting in approximately 15 minutes of down time for the network card.

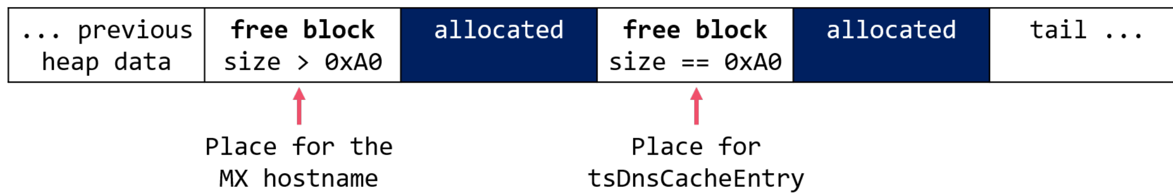

Figure 4.2: Target shape

buffer of any size with arbitrary alpha-numerical contents, along with a less-controlled `addrinfo` struct.

However, this allocation primitive alone is not sufficient as every allocated `addrinfo` and the name buffers associated with it are freed after DNS parsing fails. If DNS parsing doesn't fail, no further MX requests will be sent, preventing us from sending more responses. In order to create holes of arbitrary size, we need both an allocation primitive that will get freed (the hole) and another allocation that won't, preventing coalescing.

4.4.3 Persistent Allocation Primitive (or: Poking Holes)

To put allocated "seperators" between our newly created holes, we need an allocation that won't get freed between different DNS request-response pairs. A memory leak in the DNS parsing loop comes to the rescue. For most record types, parsing begins like this:

```
// First, check if the type fits for A resource records
if (t_dns_cache_entry.dnscQueryType == DNS_TYPE_A && answer_rr_dns.type == DNS_TYPE_A) {
  // Make sure RDLENGTH is sane for this type - for A records, 4 bytes of IP
  if (answer_rr_dns.RDLENGTH == 4) {
    new_addr_info = tfDnsAllocAddrInfo();
    // Further parsing...
    // Linking the new addrinfo to the list (t_dns_cache_entry.dnscAddrInfoPtr)
  } else {
    // Exit with error code
  }
}
```

However, in the case of MX records, the order in which validity checks are applied is different:

```
// Check if the type fits for MX resource records
if (t_dns_cache_entry.dnscQueryType == DNS_TYPE_MX && answer_rr_dns.type == DNS_TYPE_MX) {
  new_addr_info = tfDnsAllocAddrInfo();
  if (new_addr_info != NULL) {
    if (answer_rr_dns.RDLENGTH >= 2) {
      // Further parsing...
      // Linking the new addrinfo to the list (t_dns_cache_entry.dnscAddrInfoPtr)
    } else {
      // Exit with error code
    }
  }
}
```

According to this logic, if an MX resource record with `RDLENGTH` field smaller than 2 is supplied, then a new `addrinfo` struct will be allocated but the function will finish execution before linking it to the cache entry. Since DNS parsing failure results only in the cache entry and all subfields - including the `addrinfo` list - being freed, this new unlinked `addrinfo` will not be freed, resulting in a memory leak. We can craft a packet containing a few valid MX records to allocate a buffer of arbitrary size, concluding with an invalid MX record that will place an

JSOF

unlinked addrinfo at the end of our buffer. Upon DNS parsing exit, all valid MX names and addrinfo structs will be freed except for the unlinked addrinfo, resulting in a hole pattern as described above.

4.5 Overwriting a Far Call Destination

After reliable overflow of tsDnsCacheEntry.dnscAddrInfoPtr is achieved, we need to pick a destination address to write our CNAME buffer pointer to. Since our overflow allows alpha-numerical values (including "." and "-") only, our accessible address space is rather limited. Unfortunately, none of the loaded firmware files (bootloader, OS, application) are loaded in an alpha-numeric accessible address. However, we have a few assisting factors. First, the last byte of our overflow, as common in string overflows, is a null byte. In little-endian architectures, the the last byte is the most significant one - in our case, segment high byte. Second, because of the weird way segmentation works on this processor (8-bit shift instead of 4-bit), we can reach non-alphanumeric segments relatively easily.

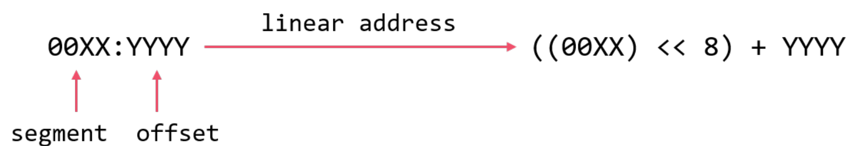

Figure 4.3: Segmented address to linear address calculation

As apparent from figure 4.3, it is possible to pick XX and YY values that are alphanumeric, yet their sum contains a high non-alphanumeric byte.

It turns out that most of the code handling heap operations (free, malloc, etc.) is relocated dynamically to segment 0x008C. This segment is reachable using the method described above.

One of the ways to achieve code execution by writing a pointer to a code section address is to overwrite a far call opcode. In x86 16-bit far calls are encoded as following:

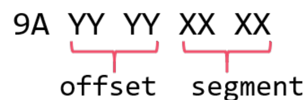

Figure 4.4: x86 far-call opcode

Our far pointer write fits the offset-segment pattern, so we can change the destination of any far-call in an alpha-numerical address to point to our controlled CNAME buffer.

The far-call we chose is located in an error flow in free which originally calls assert. We specifically chose this far-call destination for two reasons. First, this error flow will be executed when the MX record from which our overflow happened. This allows us to control the shellcode execution. Second, in this error flow, the stack frame will contain a pointer to the end of the CNAME buffer that contains our alphanumeric shellcode. As will be explained later on, when running an alphanumeric shellcode, a pointer to a location in the shellcode is necessary to achieve arbitrary shellcode execution (rather than alphanumeric only).

4.6 Executing Shellcode

Building on previous stages, we can now assume there is a malicious far-call in the free() error flow. This flow will be triggered when the prefix and postfix size fields of the block passed to

free() do not match (see section 4.2.2). This is necessarily true for the MX name buffer from which the overflow happened: because the overflow is linear, it will always overwrite the postfix size field, but not the prefix size field. This mismatch will cause the error flow to take place, executing our far-call and redirecting execution to the buffer containing our arbitrary CNAME.

4.6.1 Triggering tsDnsCacheEntry Free

The free() error flow will be executed only when the cache entry is freed. This won't happen automatically, even though the DNS TTL can be specified to be zero because the cache-entry TTL is checked and subsequently freed only when another DNS request for the same domain is generated by the device. DNS cache entry freeing can be remotely triggered by causing system events that trigger MX requests, such as attempting to login to the Web/FTP/SSH/Telnet interfaces with bad credentials. Freeing should also take place without manual interference due to regular system events (routine system checks, power fluctuations, etc).

4.6.2 Executing Arbitrary Shellcode

Reaching arbitrary shellcode execution from alpha-numeric shellcode on x86 is a well-known problem and has been discussed in depth and solved long ago. 16-bit mode makes this a bit trickier, but the essence is the same. We used a technique inspired by [1], in which a two-stage decoder is implemented:

1. The first decoder is completely alpha-numerical - using push/pop instructions to move values between registers, and xor against memory in the cs segment to modify itself (self-modifying). An alpha-numeric only shellcode decoder is difficult to write without having a pointer to a known position in the shellcode. In the execution context we chose the free() assert error flow because this way, when the shellcode starts running there is a pointer on the stack to the end of the CNAME buffer, and this satisfies our prerequisite for a pointer in to the shellcode. We used pop to move this pointer in to a register and then use it to xor other parts of the shellcode, generating non-alphanumeric opcodes. This stage also pushes the encoded arbitrary shellcode on the stack before handing over execution to the next stage. One of the requirements for DNS names is having a "." character at least every 63 characters, but luckily in x86 this is the cs: prefix for memory operations, which is needed anyway for this stage of decoding.
2. This stage is hard-coded and not arbitrary, to make writing the alpha-numerical decoder from the previous stage easier (alpha-numerical x86 doesn't allow complex flow control). However, it is non-alpha-numerical, allowing for more complex code structures such as loops, arbitrary memory manipulation and arbitrary jumps & calls. This stage loops over and decodes the encoded alpha-numeric code we pushed to the stack in the last stage, finishing by handing over execution to it.

When these decoders are done, we can run any payload without worrying about encoding issues. Our payload of choice is rather simple, as it just turns off the UPS power outlets by calling a built-in function, but any other payload of course can be executed as well.

4.7 Putting It All Together

To recollect, the exploit consists of four main stages:

1. **Crashing the device** by waiting for any DNS request and corrupting the heap. This is done to reach a deterministic heap state.

2. **Shaping the heap** by replying to the first 2 MX requests with resource records of carefully chosen sizes, followed by a corrupted MX resource record that will cause a memory leak, securing our hole.

3. **Overflow** `tsDnsCacheEntry` via the 3rd MX request, resulting in a pointer to a controlled buffer being written as a far call destination.

4. **Trigger** or wait for an event that causes a DNS request to be issued and our `tsDnsCacheEntry` to be freed. Execution will be redirected through the corrupted far-call to our alphanumeric shellcode. This results in a two-stage decoding procedure being executed leading to arbitrary shellcode execution from the stack.

Chapter 5

References

[1] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004. ISBN 0764544683.

[2] P. V. Mockapetris. Domain names - implementation and specification. *RFC*, 1035:1–55, 1987. doi: 10.17487/RFC1035. URL https://doi.org/10.17487/RFC1035.

[3] S. Oberman and M. Kol. CVE-2020-11896 white paper. 2020. URL https://www.jsof-tech.com/ripple20.

WHO WE ARE

JSOF is a multidisciplinary team focused on solving product cyber security challenges. We are research oriented and focus exclusively on product security. We excel in projects that are complex, time-sensitive, or mission-critical.

CONTACT US

www.jsof-tech.com
info@jsof-tech.com

JSOF

Ripple20 - © 2020 JSOF Ltd.