CVE-2020-11896 RCE
CVE-2020-11898 Info Leak

Moshe Kol, JSOF
Shlomi Oberman, JSOF

JSOF

JUNE 2020

Ripple20 - © 2020 JSOF Ltd.

Abstract

JSOF research lab has discovered a series of multiple zero day vulnerabilities in the Treck TCP/IP stack that is widely used in embedded and IoT devices.

In this document we will be giving some background on Ripple20 and the Treck TCP/IP stack and explain the details and exploitation of two of the vulnerabilities found in the IP fragmentation component, both result from the same root cause.

One of these vulnerabilities is an RCE and the other an information leak. We developed a proof-of-concept exploit on a Digi Connect ME 9210 device, which is an Ethernet system-on-module embedded in to other devices as a communication component. Any device embedding this component, or any of a series of other components from the same line, are probably affected by the same vulnerabilities and can be exploited with the same techniques.

A second white-paper, to be released following BlackHat USA 2020, will be detailing the exploitation of CVE-2020-11901, a DNS vulnerability, on a Schneider-Electric APC UPS device.

Contents

Ripple20 - © 2020 JSOF Ltd.

Chapter 1

What is Ripple20?

Ripple20 is a series of 19 zero day vulnerabilities that affect hundreds of millions of devices, and were discovered by JSOF research labs in a TCP/IP stack that is widely used in embedded and IoT devices.

The starting point for these vulnerabilities is an embedded TCP/IP low-level Internet protocol suite library by a company called Treck, inc. This is a basic networking element, a building block, useful in any context, for any device that works over a network.
Treck TCP/IP is a proprietary fully featured TCP/IP communication stack designed for embedded devices and real-time operating systems. While our research was focused on security, our general impression is that it works well and has high performance. The large client base would suggest the same.

In their own words:

> *"Treck TCP/IP is designed to be high performing, scalable and configurable for easy integration into any environment regardless of processor, commercial RTOS, proprietary RTOS, or if no RTOS is being used. By designing API's to interface the kernel, timer, driver, sockets the Treck TCP/IP provides easy integration into a wide variety of embedded products."*

The damaging effects of a these vulnerabilities has been amplified like a ripple effect to a dramatic extent due to the supply chain factor. A single corrupt component, though it may be relatively small, impacts a wide range of industries, applications, companies, and people.

The vulnerabilities have a wide affect on entire supply chains in almost all sectors in the IoT and embedded device world. The vendors that produce the affected device range from one-person boutique shops to fortune 500 multinational corporations, and from connected smart homes to industrial and medical device.

Of these vulnerabilities:

- 4 are critical remote code execution vulnerabilities with CVSS ≥ 9
- 4 are major with a CVSS ≥ 7
- 11 more have various lower severity, information leaks, DoS and others

[Of the 19 vulnerabilities, 2 were reported anonymously and the rest were reported by JSOF.]

We named the vulnerabilities Ripple20 to capture the ripple effect caused by these vulnerabilities, the "20" added at the end has multiple meanings for us:

Ripple20 - © 2020 JSOF Ltd.

- The vulnerabilities were reported in 2020.
- The Treck stack has been around for more than 20 years. Possibly the vulnerabilities too.
- There are 19 vulnerabilities, and just like the candles on a birthday cake we are adding one for next year!

Ripple20 - © 2020 JSOF Ltd.

Chapter 2

CVE-2020-11896 Overview

CVE-2020-11896 is a critical vulnerability in Treck TCP/IP stack. It allows for Remote Code execution by any attacker that can send UDP packets to an open port on the target device. A prerequisite of this vulnerability is that the device supports IP fragmentation with IP tunneling. In some of the cases where this prerequisite is not met, there will remain a DoS vulnerability.

2.1 Preliminary Networking Concepts

To understand this vulnerability we need to familiarize ourselves with two networking concepts: IPv4 fragmentation and tunneling.

2.1.1 IPv4 Fragmentation

IP fragmentation makes it possible an IP packet to be sent in a network even in cases where its size is larger than the maximum size allowed in a specific link of a network. IP fragmentation is a technique where the packet is split in to several smaller parts ("fragments") to support transmission over those links and networks. The protocol supports fragmentation and then reassembly of th packets. IPv4 fragmentation is described in [1].

Different fragments are grouped using an *identification* field in the IP header. This identification field describes which packet a fragment belongs to. This allows for different packets to travel fragmented through the network and be reassembled correctly on the other side. The last fragment has the `MF` (More Fragments) bit flag set to 0, where all other fragments have `MF=1`.

It is the responsibility of the network stack to fragment a large packet and send the multiple fragments over the network. An example would be a UDP application which asks to send a large datagram. It is also the responsibility of the network stack to reassemble a fragmented packet when it is received.

If only part of the fragments of a packet arrive, the network stack will eventually abandon these fragments. In most implementations, when any fragment is processed, the network stack starts a timer. When this timer elapses, the network stack discards all the fragments which belong to the same *identification* group. This is done to prevent overloading of the device memory.

2.1.2 IP Tunneling

IP tunneling allows for virtual point-to-point links between two separate networks. It is achieved by encapsulating a packet (which may be an IP packet) within another packet, so that the inner packet has a different source and destination addresses than the outer packet.

You can think of it as an international package shipment where the package is labeled with the full source and destination address in the respective countries, but when the package is

Ripple20 - © 2020 JSOF Ltd.

Figure 2.1: Illustrating tunneling

shipped overseas it is placed in a container (like a ship or plane) that only has the address of the . destination and source port

The source and destination addresses of the outer packet are the tunnel-endpoints, and the addresses in the inner packet are used for network routing on both ends of the tunnel.

A tunnel entry-point is the node that receives an IP packet that should be forwarded over the tunnel. It encapsulates this packet within an outer IP packet. When the packet arrives at the tunnel exit-point, it is decapsulated and forwarded as if it were a regular packet sent in the target network.

A prime example of tunneling usage is in Virtual Private Network (VPN) technologies.

There are several tunneling protocols, one of the most simplest and oldest is IP-in-IP (IP protocol number 4).

IP-in-IP

IP-in-IP is an IP tunneling protocol where one IP packet is encapsulated in another IP packet by adding an outer IP header with source and destination addresses that are equal to the tunnel's entry-point and exit-point, respectively.

The inner packet is unmodified, and the outer IP header copies some fields from the inner IP header. The outer header has IP protocol number 4. More information about IP-in-IP tunneling can be found in [5] and [2].

2.2 Treck TCP/IP Stack Internals

Packets in the Treck TCP/IP stack are represented by a structure called `tsPacket`. Each packet is associated with a data buffer that holds the raw data that arrived from the interface driver. The `tsPacket` struct also holds another important structure called `ttUserPacket`, and a pointer to a `tsSharedData` structure, which includes information that the network stack needs when it processes the packet (pointer to a socket structure, src/dst addresses or ports, etc).

The `tsPacket` structure contains several fields involved in the vulnerabilities described below. For this reason it is important to understand this structure before continuing.

```
1  struct tsPacket {
2      ttUserPacket pktUserStruct;
3      ttSharedDataPtr pktSharedDataPtr; // Point to corresponding sharable ttSharedData
4      struct tsPacket * pktChainNextPtr; // Next packet (head of a new datagram in a queue)
5      struct tsDeviceEntry * pktDeviceEntryPtr; // pointer to network Device struct
6      union anon_union_for_pktPtrUnion pktPtrUnion;
7      tt32Bit pktTcpXmitTime;
8      tt16Bit pktUserFlags;
9      tt16Bit pktFlags;
10     tt16Bit pktFlags2;
```


Ripple20 - © 2020 JSOF Ltd.

```
    tt16Bit pktMhomeIndex;
    tt8Bit pktTunnelCount; // Number of times this packet has been decapsulated. Initially set to zero.
    tt8Bit pktIpHdrLen; // Number of bytes occupied by the IP header.
    tt8Bit pktNetworkLayer; // Specifies the network layer type of this packet (IPv4, IPv6, ARP, etc).
    tt8Bit pktFiller[1];
};
```

and this is the contained `ttUserPacket` structure (typedef of `tsUserPacket`):

```
struct tsUserPacket {
    void * pktuLinkNextPtr; // Next tsUserPacket for fragmented data
    ttUser8BitPtr pktuLinkDataPtr; // Pointer to data
    ttPktLen pktuLinkDataLength; // Size of data pointed by pktuLinkDataPtr
    ttPktLen pktuChainDataLength; // Total packet length (of chained fragmented data). Valid in first link only.
    int pktuLinkExtraCount; // Number of links linked to this one (not including this one). Valid in first link only.
};
```

Both these structures are built to support IP fragmentation and so we will describe some fields which relate to fragmentation. We could think of non-fragmented packet as a special case of fragmentation where there is only one fragment.

The field `pktuLinkDataPtr` points into the data buffer of the current fragment. The exact location within this data buffer changes as the network stack processes the packet in different stages and depends on the layer of the packet which is currently being processed. For instance, when the network stack process the Ethernet layer (in `tfEtherRecv`), this field points to the Ethernet header.

The `pktuLinkDataLength` field specifies the size of the data pointed to by `pktuLinkDataPtr`, i.e., *the size of a single fragment*.

The `pktuLinkNextPtr` is used to keep track of the fragments in the packet. This field points to another `tsPacket` which represents the next fragment, which in turn also holds a reference to the next fragment and so forth. For this reason we can also refer to fragments as "links" in this linked list. If this link is the last fragment, or if the data is not fragmented, this field will equal NULL.

The `pktuChainDataLength` field represents the packet length including all of the fragments, i.e., *the total size of the packet*. It is set only for the first fragment, and if the data is not fragmented it is equal to `pktuLinkDataLength`.

A common pattern in the stack is to adjust the `pktuLinkDataPtr` pointer when moving between layers in the stack. For instance, if our packet is an ICMP echo-request packet (ping), it will be composed of three layers: Ethernet, followed by IPv4, followed by ICMP. In this case, when the Ethernet layer is processed (in the function `tfEtherRecv`), the `pktuLinkDataPtr` points to the start of the Ethernet header, and then before moving to the next layer, it is adjusted using the following code:

```
pkt->pktuLinkDataPtr = pkt->pktuLinkDataPtr + 0xe;
pkt->pktuLinkDataLength = pkt->pktuLinkDataLength - 0xe;
pkt->pktuChainDataLength = pkt->pktuChainDataLength - 0xe;
```

In this case, 0xe (14 in decimal) is the size of the Ethernet header (6 (dst MAC) + 6 (src MAC) + 2 (etherType)).

When `tfEtherRecv` finishes with packet processing, it forwards the packet to next layer of processing using the EtherType field that represents the next layer's protocol. Supported EtherTypes encountered are ARP, IPv4 and IPv6.

Ripple20 - © 2020 JSOF Ltd.

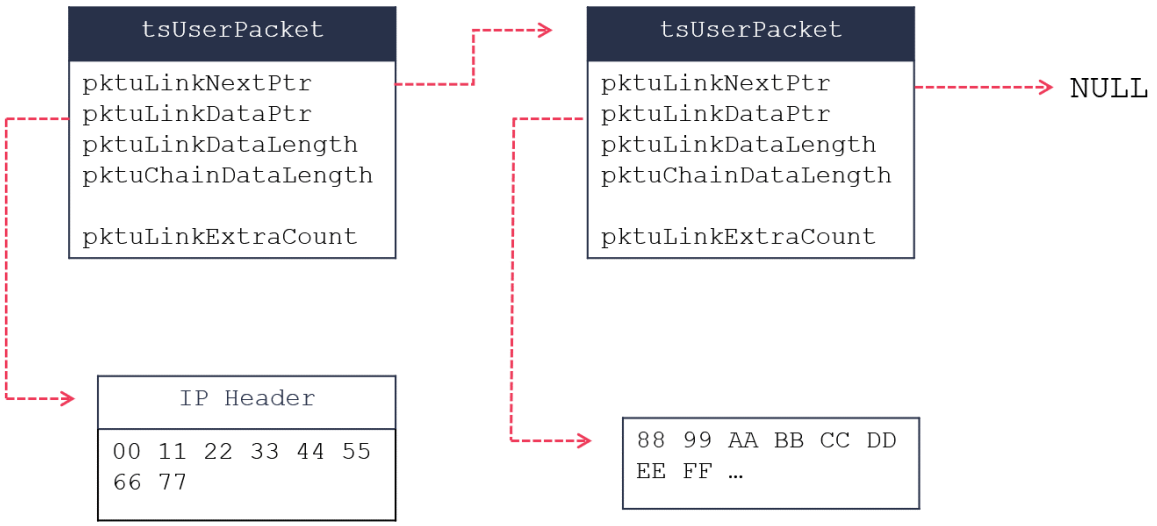


Figure 2.2: Packet with fragments

In our example, when the IPv4 layer receives the packet (in the function `tfIpIncomingPacket`), the pointer `pktuLinkDataPtr` already points past the Ethernet header, so it can safely assume that the data pointed to by `pktuLinkDataPtr` is an IPv4 header.

Incoming data is processed by functions with the same naming convention `tf*IncomingPacket` (as far as we've seen), where * is the protocol name. In the case of Ethernet/IPv4/ICMP the packet will be processed by the functions `tfEtherRecv`, `tfIpIncomingPacket`

and `tfIcmpIncomingPacket`, respectively.

The Treck stack handles the reassembly of fragments in the procedure `tfIpReassemblePacket`, called from `tfIpIncomingPacket`. This procedure is called whenever an IP fragment destined to device is received. If there are missing fragments, the function returns NULL. Otherwise, if all fragments arrive and there are no holes, the network stack links the fragments together using the `pktuLinkNextPtr` field and passes the packet for further processing in the next layer. The word "reassembly" in this context does not mean copying of the packet to a contiguous memory block, but rather simply chaining them together in a linked-list.

2.3 Vulnerability Root Cause

To understand the root cause of the vulnerability, we'll take a quick look at two fields present in the IP header:

- IHL (4 bits): the size of the IP header in dwords. Minimum value is 5 (20 bytes). If there are IP options, the header length gets larger, up to a maximum value of 0xf (60 bytes).
- Total Length (2 bytes): the size of the entire IP packet in bytes (or IP fragment, if fragmented), including the header.

The function `tfIpIncomingPacket` begins with some basic sanity checks. In addition to verifying the header checksum, it also verifies that:

Ripple20 - © 2020 JSOF Ltd.

```
ip_version == 4 &&
data_available >= 21 &&
header_length >= 20 &&
total_length > header_length &&
total_length <= data_available
```

The "data_available" is measured using the field `pktuChainDataLength`.

If all sanity checks pass, the function checks whether the specified *total length* in the IP header is strictly less than the packet's `pktuChainDataLength`, indicating that more data was actually received than is stated in the IP header. If it is true, a trimming operation takes place to remove the extra data (decompilation code reorganized for clarity):

```
1 if ((uint)ipTotalLength <= pkt->pktuChainDataLength) {
2   if ((uint)ipTotalLength != pkt->pktuChainDataLength) {
3       pkt->pktuChainDataLength = (uint)ipTotalLength;
4       pkt->pktuLinkDataLength = (uint)ipTotalLength;
5   }
6   ...
7 }
```

This is where the vulnerability lies. Recall that `pktuLinkDataLength` holds the size of the current fragment and `pktuChainDataLength` holds the size of the entire packet. If the operation above takes place, an inconsistency is created, where `pkt->pktuChainDataLength == pkt->pktuLinkDataLength`, but there might be additional fragments pointed to by `pkt->pktuLinkNextPtr`. Another way to think of this is as a fictitious and inconsistent state where the total size of the fragments on the linked list is larger then the size stored in `pktuChainDataLength`.

The inconsistency created by the weak trimming operation does not bode well for the rest of the processing. *However, we have another challenge to overcome.* The `tfIpIncomingPacket` function is called with one received fragment every time and calls `tfIpReassemblePacket` to handle it. `tfIpReassemblePacket` is responsible for inserting the fragments in to the linked list described above. It does not copy the fragments in to a continous memory buffer. When all the fragments have been received, `tfIpReassemblePacket` returns the complete packet as a linked-list of fragments for further processing on the next protocol layer. This reassembly operation is executed *after* the vulnerable trimming operation. As soon as the reassmbly operation is finished, `tfIpIncomingPacket` will either return or pass the packet forward for processing at the next netowrk layer (for example: UDP). This prevents us from performing the exploit, as we need a fragmented packet in order to reach the inconsistent state. In other words, the vulnerable code is only supposed to be executed on a per-fragment basis (or on single- fragment packets). If it is executed in this way, it is not actually vulnerable.

So – *How can we trigger the vulnerable trimming flow with incoming fragmented data so that the aforementioned inconsistency is achieved?*

2.3.1 Processing Fragmented Packet at the IP Layer

In order for the fragmented packet to be processed at the IP layer and reach the vulnerable flow, we use tunneling.

Tunneling allows for the *inner* IP packet to be processed by `tfIpIncomingPacket` as a non-fragmented packet. The function `tfIpIncomingPacket` will be called twice recursively, once for the inner layer of the IP tunneling and several times for the outer layer (once for every fragment). First, `tfIpIncomingPacket` will receive all fragments from the *outer* layer, call `tfIpReassemblePacket` on each, and once they are all received, it will pass execution to the next protocol layer, which in this case is IPv4 again, so `tfIpIncomingPacket` will be called from `tfIpIncomingPacket` with the *inner* IP layer.

Ripple20 - © 2020 JSOF Ltd.

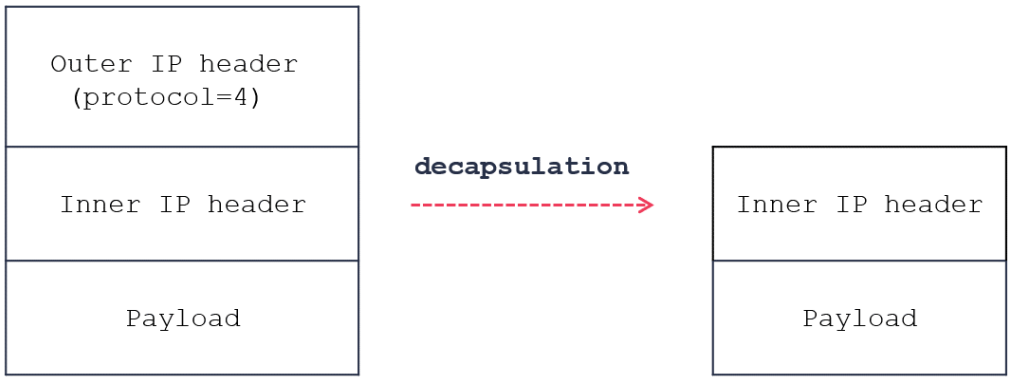


Figure 2.3: Tunnel decapsulation

Fragmenting the outer IP packet results in `tfIpIncomingPacket` being called with the inner packet that now consists of several fragments but is marked as non fragmented in the IP header (`MF=0`). In terms of the data structure describing the packet, it is now composed of several separate links in a linked-list, each with a separate `pktuLinkDataLength` value.

Let's get more concrete. Consider the following example which will accompany us throughout the paper:

- Inner IP packet: `IPv4{len=32, proto=17}/UDP{checksum=0, len=12}` with payload `'A'*1000`.
- Outer IP packet (fragment 1): `IPv4{frag_offset=0, MF=1, proto=4, id=0xabcd}` with 40 bytes from the inner IP packet as payload.
- Outer IP packet (fragment 2): `IPv4{frag_offset=40, MF=0, proto=4, id=0xabcd}` with the rest of the bytes from the inner IP packet as payload.

(We set the checksum field to 0 since that will cause the UDP checksum verification to be skipped.)

When the network stack handles the outer fragments, it links them using the field `pktuLinkNextPtr` in `tsUserPacket` structure, as already described. When the function `tfIpIncomingPacket` processes the inner IP packet (due to protocol=4), it is processing incoming fragmented data (the inner IP packet is represented by two `tsPacket` structures linked together), but will still call the vulnerable flow, thus solving our challenge.

Furthermore, the inner IP packet passes the IP header sanity checks since only the `pktuChainDataLength` field of `tsUserPacket` is taken into account (rather than `pktuLinkDataLength`). In our example, the total length of the inner IP packet (32) is smaller than the chain-data-length (1000+8+20=1028), so the Treck stack will attempt to trim the packet incorrectly, by setting both the fields `pktuLinkDataLength` and `pktuChainDataLength` to the same value - the total IP length (32 in our example). This leads to a situation where the inner IP packet is represented by two `tsPacket` structures linked together, but their total cusize is larger than the `pktuChainDataLength` field (instead of 1028 bytes, the `pktuChainDataLength` field equals 32 after trimming).

Ripple20 - © 2020 JSOF Ltd.

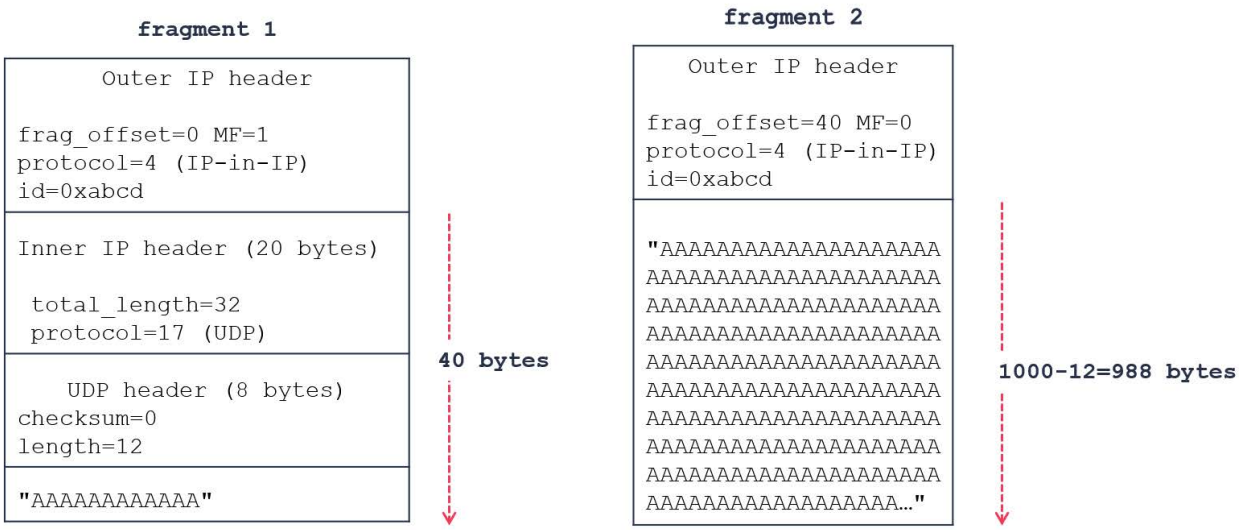


Figure 2.4: Example packets illustration

Achieving Heap Overflow using UDP 2.3.2

Now that we've reached an inconsistent state, we face another question - How can we exploit this inconsistency to gain a memory corruption primitive?

It turns out that there is at least one code path where fragmented data gets copied into a single continuous buffer. This is part of the code handling UDP datagrams. The internal logic of this flow consists of a new packet being allocated (using `tfGetSharedBuffer`), the size of which is based on the `pktuChainDataLength` field, followed by a copy of the different fragments of the packet into the newly allocated packet one by one.

The function responsible for doing the copy is `tfCopyPacket`, which accepts a source packet and a destination packet, in this order. Here's an excerpt of the fragment copy code:

```
1  i = 0;
2  do {
3      memcpy(dst->pktuLinkDataPtr + i,src->pktuLinkDataPtr,src->pktuLinkDataLength);
4      i = i + src->pktuLinkDataLength;
5      src = (tsPacket *)src->pktuLinkNextPtr;
6  } while (src != NULL);
```

As you can see, the function `tfCopyPacket` doesn't take into account the length of the buffer it writes into. It simply takes every link from the `src` packet (our fragmented packet) and copies its data into the destination packet. The destination packet was allocated based on the `pktuChainDataLength`, and so if the vulnerability was previously triggered, then the allocated buffer may be smaller than the sum of all the individual links in the packet after our invalidation - hence, a heap overflow occurs.

One thing that remains to be described is how we trigger this flow.

If an application is listening on any UDP port, a UDP packet destined to this port is passed to a socket handling function `tfSocketIncomingPacket`. Its job is to append that packet to the socket receive queue (which is later polled by the application layer).

Ripple20 - © 2020 JSOF Ltd.

In our research, we found that when the socket receive queue for UDP packets is non-empty and a new packet arrives, the flow described above containing the heap overflow is achievable. Take a look at this excerpt of `tfSocketIncomingPacket`:

```
local_10 = pkt;
/* ... */
if (sockEntry->socReceiveQueueNextPtr == NULL) {
    sockEntry->socReceiveQueueNextPtr = pkt;
    queueLastPtr = local_10;
}
else {
    queueLastPtr = sockEntry->socReceiveQueueLastPtr;
    if ((queueLastPtr->pktSharedDataPtr->dataFlags & 0x40) == 0) {
        /* Shared data doesn't point to user device memory */
        sizeOfPacketBuffer = (uint)(pkt->pktSharedDataPtr->dataBufLastPtr +
        ↪  -(int)pkt->pktSharedDataPtr->dataBufFirstPtr);
    }
    if (protoNum == 6) {
        /* Related to TCP; redacted for brevity  */
    }
    else {
        if (sizeOfPacketBuffer != 0) {
            uVar2 = (uint)sockEntry->socRecvCopyFraction * pkt->pktuChainDataLength;
            if (uVar2 <= sizeOfPacketBuffer) {
                dst = tfGetSharedBuffer(0x54,pkt->pktuChainDataLength,0);
                if (dst != NULL) {
                    tfCopyPacket(pkt,dst);
                    needToDrop = true;
                    local_10 = dst;
                }
            }
        }
        queueLastPtr->pktChainNextPtr = local_10;
        queueLastPtr = local_10;
    }
}
```

We see that in order to reach `tfGetSharedBuffer` we need to circumvent a check involving `socRecvCopyFraction`. We don't know its exact purpose, but through debugging and experimentation we found that its value is 4 (in our case).

In our recurring example, our first packet link has small buffer size so `sizeOfPacketBuffer` is relatively small (on the order of 10s of bytes).

But when we reach this flow, `pkt->pktuChainDataLength` equals 4 (after trimming it was 32, then decremented by 20 (size of IP header) when processing the IP layer, and then decremented again by 8 (size of UDP header)). So 4*4=16 is less than `sizeOfPacketBuffer`, and we pass this check.

One last thing that we need to make sure of is that the receive queue for UDP packets is non-empty (otherwise this flow is not reached). In thoery there could be several ways to do this. In our exploitation, we found that sending many UDP packets to the same port fast enough would do the trick. However, getting this part to work reliably was tricky. The exploit is written in Python using `Scapy` that turns out to be too slow for our purpose. To overcome the obstacle, we used the `L3socket` object of `Scapy` and instantiated a bunch of threads that only flood the device with benign UDP packets, so that the socket receive queue will be non-empty. Writing the code in C or Go would probably also work. Additional improvments can be made to this part, depending on the device being exploited and the listening server.

Another obstacle is that before we arrive at `tfSocketIncomingPacket`, which is where the overflow occurs, our vulnerable packet passes through `tfUdpIncomingPacket`. This function contains some sanity checks related to UDP, so we need to pass those as well:

Ripple20 - © 2020 JSOF Ltd.

```
udpLen = udpHdr->udpLength >> 8 | udpHdr->udpLength << 8;
if ((udpLen < 8) || (*(ushort *)&pkt->pktuChainDataLength < udpLen)) goto dropPacket;
if (udpLen < *(ushort *)&pkt->pktuChainDataLength) {
    tfPacketTailTrim(pkt,(uint)udpLen,0);
}
/* ... */
if ((udpHdr->udpChecksum != 0) && (tvUdpIncomingChecksumOn != 0)) {
    /* Compute checksum... */
}
```

As we can see, we can avoid this type of trimming (not to be confused with the vulnerable flow), by ensuring that the UDP length field equals the `pktuChainDataLength` field minus the size of the inner IP header.

To summarize: if a UDP port is listening on our device, we can quickly send packets so that the socket receive queue will be non-empty. At the same time, we will send a fragmented UDP packet that triggers the vulnerability and ticks a few checkboxes. The result we expect is a small buffer allocated on the heap with `tfGetSharedBuffer` which is then overflown by `tfCopyPacket`.

Ripple20 - © 2020 JSOF Ltd.

Chapter 3

Exploiting CVE-2020-11896 on Digi Connect ME 9210

We used a Digi Connect ME 9210 device along with a development board for proof-of-concept exploit development and achieved remote code execution.

This device looks like an Ethernet connector, and is actually a small "ultra-compact" system-on-module. Its longest dimension is 3.67cm and it comes fully featured with a built-in CPU, operating system, and of course - networking stack. The module can be programmed and its firmware can be customized. It is presumably used for offloading of networking capabilities as well as advanced features like a built-in hardware encryption engine and multiple interface options like CAN-BUS.

This is an interesting device because it is highly embeddable, and is used in critical settings, like medical devices for example. This device can also be purchased from any of the large electronic-parts re-sellers. This makes vulnerabilities extremely dangerous as they could potentially affect any device embedding these modules and it is difficult to know which devices use it. The device version we tested is the "NET+OS" version. There are different versions and we don't know if they are affected.

We chose this device as a first device to research and exploit because of these reasons, as well as the lesser complexity of exploiting with a full setup including development board, JTAG debugger, symbols(!) etc.

Our proof-of-concept exploit runs a shellcode which makes a LED on the development board blink indefinitely. We will now describe our exploit - the primitives established and the techniques used. The exploit is probably easily adaptable to devices using the Treck stack that use the same heap configuration and a similar (slightly older) version of Treck.

3.1 Exploit Strategy

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

- Override heap meta-data information.
- Overflow application-specific data structures.

The former is generally considered more generic, as it does not require any specific data structure to be present, only some memory blocks, possibly with a specific size.

As the Treck stack is highly configurable, and used in many settings, and since the heap structure seems to be very suitable for exploitation with no mitigations present, we chose to override heap meta-data information and not rely on any specific features enabled or added by Digi.

Ripple20 - © 2020 JSOF Ltd.

3.2 Treck Heap Internals

On this device the Treck TCP/IP stack uses a proprietary heap. Let's have a look under-the-hood.

The following description concerns the heap implementation found in Digi. Another heap mechanism implemented by Treck is called Sheap (Simple Heap). The use of Sheap must be enabled during compilation, and in the case of Digi, it wasn't enabled. Treck can also use a user-supplied heap and possibly supports other options. Note also that this device has an older version of the Treck stack, and newer versions might have different implementations.

The heap uses several fixed-size allocation buckets, each tracked using a singly-linked list. There is a separate component which handles allocation requests of sizes above 0x1000 ("large allocations").

These fixed-size allocation buckets are referred to by Treck as `QX`, `Q0`, `Q1`, `Q2`, `Q3`, `Q4`, `Q5` and `Q6`. The following table describes each bucket, its block size and the allocation size it corresponds to:

Bucket	Block Size	Allocation Size
QX	32 (0x20)	1 - 32
Q0	64 (0x40)	33 - 64
Q1	128 (0x80)	65 - 128
Q2	256 (0x100)	129 - 256
Q3	512 (0x200)	257 - 512
Q4	1024 (0x400)	513 - 1024
Q5	2048 (0x800)	1025 - 2048
Q6	4096 (0x1000)	2049 - 4096

Each bucket list serves as a free blocks cache for its associated block size. When a block is freed, its relevant bucket is determined (using the heap metadata) and it is inserted at the head of the list.

The Treck stack allocates memory for data buffers and structures mainly using `tfGetRawBuffer`. This function receives a 4-byte size and returns a pointer to an allocated memory block we will call a "raw buffer", or NULL if it fails. A raw buffer is freed using the function `tfFreeRawBuffer`.

If the allocation request size is small (less than 0x1000), the appropriate free blocks cache is consulted:

```
/* the rawNextPtr field is at offset 0 */
mem = bucketQueue->rawNextPtr;
if (mem != NULL) {
    /* remove from head of bucket queue */
    bucketQueue->rawNextPtr = mem->rawNextPtr;
    /* ... */
    mem->rawNextPtr = NULL;
}
/* ... */
if (mem != NULL) {
    return (ttRawBufferPtr)mem;
}
```

If `mem == NULL` at the end of the cache lookup, a new block is allocated from a separate allocator and returned:

```
newMemoryBlock = (uint *)tfBufferDoubleMalloc(sz + 4);
if (newMemoryBlock == NULL) {
    return NULL;
```


Ripple20 - © 2020 JSOF Ltd.

```
}
*newMemoryBlock = sz;
/* ... */
return newMemoryBlock + 1;
```

For small allocations, `sz` refers to the size of the bucket that matches the allocation request size. For large allocations, `sz` is the original allocation request size, rounded-up to the next 4-byte boundary. Note that `sz` is saved on the heap as metadata.

Treck's function `tfBufferDoubleMalloc` doesn't add meta-data and internally calls `tfKernelMalloc`. In turn, `tfKernelMalloc` saves the requested size as meta-data and calls the operating system `malloc` function. The reason for the "double" in `tfBufferDoubleMalloc` is the two attempts it makes calling `tfKernelMalloc` (the second of them in the case that the first fails).

When the heap is initially setup and the initial buckets built, the same flow is used for each block - each block is allocated separtely by calling `tfBufferDoubleMalloc`, and so every block is actually allocated separately over several layers of allocators. We are not sure why this happens, and whether this is an artifact of Treck or Digi code.

In total, three instances of meta-data are saved for a single memory block allocated using `tfGetRawBuffer`:

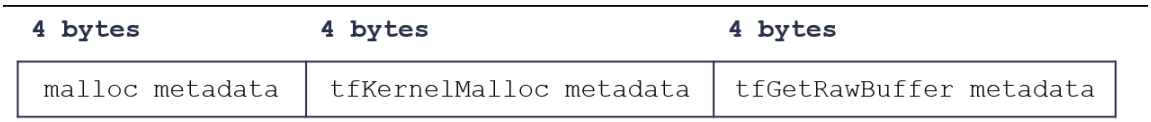


Figure 3.1: Heap metadata

For example, for an allocation request size of 0x9c, the `Q2` bucket will be selected (block size : 0x100). The raw buffer, along with its meta-data, will look like

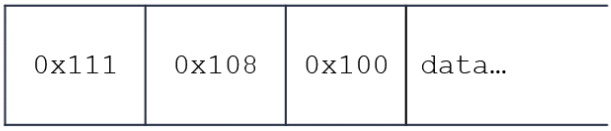


Figure 3.2: Heap metadata example

When blocks are freed using `tfFreeRawBuffer`, the (third) block size is read from the heap meta-data to determine the allocation size (0x100 in the example above). If the allocation is large, the meta-data is removed and the memory block is passed to `tfKernelFree`. For small allocations, the relevant bucket queue is determined, and the free block (`memoryBlockPtr`) is inserted at the head of the free list:

```
/* insert at the head of the free-list */
memoryBlockPtr->rawNextPtr = bucketQueue->rawNextPtr;
bucketQueue->rawNextPtr = memoryBlockPtr;
```

As you can see, the pointer to the next free memory block in the same bucket is stored on the heap as the first dword after the meta-data saved by `tfGetRawBuffer`. That is, a free raw buffer looks like this:

Ripple20 - © 2020 JSOF Ltd.

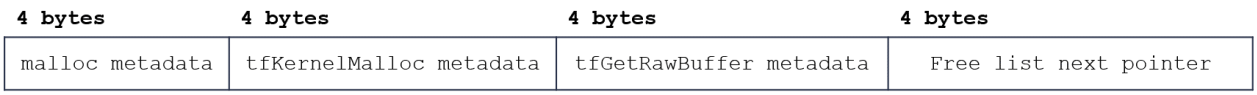


Figure 3.3: Free raw-buffer illustration

Exploitation Technique 3.3

Considering the structure of the heap and our overflow primitive, we can hopefully override the `next` pointer of a free raw buffer. Then if we manage to allocate two raw buffers from the bucket we overrided (or three, if the overflowing buffer is freed), the last raw buffer will be allocated at a location we control.

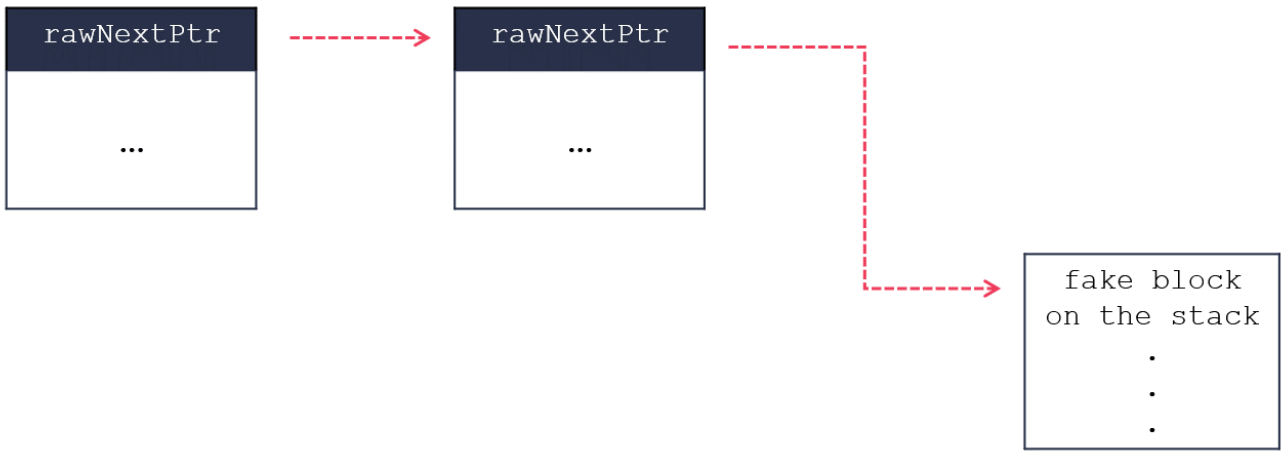


Figure 3.4: Overwriting free-list `rawNextPtr` pointer

This embedded device runs an old ARMv9 CPU and has no meaningful exploit mitigations. This means that we can hopefully allocate a raw buffer on the stack with controlled data and run a shellcode from there. We might also write a ROP chain before we redirect execution to the shellcode, so that it can perform some preliminary setup. The setup mostly consists of invalidating cache so that our shellcode will run, since otherwise the packet allocated on the stack may not be written out to memory by the time we try to execute it.

You might ask yourselves at this point - why not directly allocate a packet on the code section? The answer is that we could, but it is not necessarily simpler, and we might still run into a cache coherency issue.

Let's put this theory into practice.

3.4 Some Preliminary Shaping (or: How Packets are Allocated?)

Packets, along with their data buffer, are allocated on the heap using a function called

Ripple20 - © 2020 JSOF Ltd.

`tfGetSharedBuffer`. This function accepts three arguments: `hdrSize`, `dataSize` and a `flag`. It returns a pointer to an initialized `tsPacket` structure. We already saw this function used in `tfSocketIncomingPacket` when we described the vulnerability root cause.

The function `tfGetSharedBuffer` first allocates a `tsPacket` structure. This structure is allocated by calling the function `tfRecycleAlloc`. This function maintains a cache for highly used structures used throughout the network stack. More specifically, it is used to allocate the following fundamental structures for the Treck stack: `tsPacket`, `tsTimer`, `tsRteEntry`, `tsSocketEntry` and `tsTcpVect`. Possibly more structs could be added.

Provided that the `tsPacket` allocation succeeds, `tfGetSharedBuffer` allocates a buffer using `tfGetRawBuffer` sized roughly `sizeof(struct tsSharedData) + dataSize + hdrSize`. In unusual situations, few more bytes are added, nothing too significant. This raw buffer stores two different things consecutively - first the `tsSharedData` structure and then the continuous data buffer that stores the packet data.

Through experimentation, we found that the `tsPacket` structure is usually allocated immediately after its associated data buffer (which stores shared data and packet data). This could pose a problem for our exploitation technique, because we don't want to override a `tsPacket` structure.

To understand how we can solve this issue, let's take a quick look at `tfRecycleAlloc`:

```
size = tlRecycleArray[recycleIndex].recySize;
m = ((ttRcylPtr)tlRecycleArray[recycleIndex].recyListPtrPtr)->rcylNextPtr;
if (m == NULL) {
    m = (tsRecycleEntry *)tfBufferDoubleMalloc(size);
    if (m == NULL) {
        return NULL;
    }
    /* ... */
} else {
    /* unlink from head */
    ((ttRcylPtr)tlRecycleArray[recycleIndex].recyListPtrPtr)->rcylNextPtr = m->rcylNextPtr;
    /* ... */
}
/* ... */
return m;
```

As we can see - if the recycle list is empty, a new buffer is allocated using our old acquaintance `tfBufferDoubleMalloc`. Thus, in order to overcome our previously stated challenge, we can cause `tfRecycleAlloc` to be called enough times. This makes the "recycle array" grow in size, and any future allocations of `tsPacket` structures will be allocated in the recycle array, in a memory area "behind us", making sure we don't overflow this struct. We need to do this quickly: after a packet is processed, it is dropped and its underlying `tsPacket` structure is freed (using `tfRecycleFree`).

In summary: We need to flood our device with packets, so that many `tsPackets` will be allocated relatively at the start of the heap. Then, whenever a new packet is allocated using `tfGetSharedBuffer`, the request for `tsPacket` structure will be serviced from the recycle list (there are plenty of `tsPacket` structures in the list now). This allow us to proceed with overflowing free raw buffers.

3.5 Choosing an Allocation Size

Recall the example packets described earlier when looking at the vulnerability root cause. If we manage to trigger our vulnerable flow in `tfSocketIncomingPacket`, the following line will be triggered with `pkt->pktuChainDataLength == 4`:

Ripple20 - © 2020 JSOF Ltd.

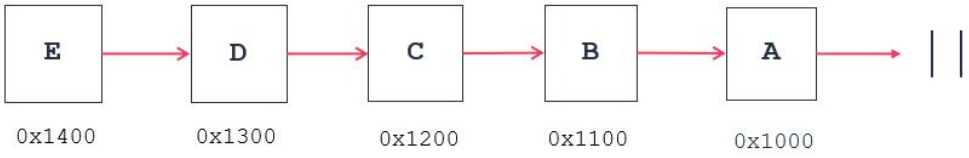
```
tfGetSharedBuffer(0x54,pkt->pktuChainDataLength,0);
```

One of the side effects of this call is the allocation of a raw buffer from the `Q2` bucket (block size 0x100).

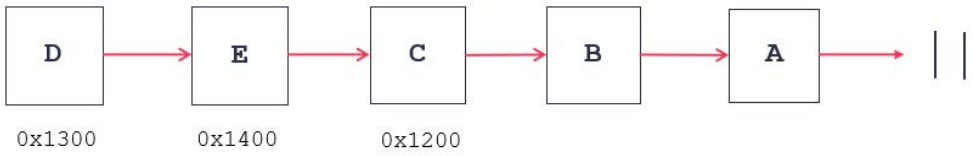
Knowing how the heap operates, if there is a free raw buffer directly after our allocation we will be able to override the `next` pointer in that block, thus corrupting the free-list.

The easiest way to shape the heap in such a way is to find some primitive that we can use to allocate raw buffers from the same bucket `Q2` (block size 0x100). Such a primitive can hopefully be used to allocate several contiguous raw buffers of size 0x100. We might need to reverse the order of the first two allocations, depending on the order these allocations are freed and considering that the list operates in LIFO mode.

To illustrate the point, consider the following hypothetical scenario. We allocate 5 blocks of size 0x100, named A-E. Block A is allocated first at address `0x1000`, block B is allocated right after block A, at address `0x1100` and so on. It so happens that the first allocated block (block A) is also the first block to be freed. After the 5 blocks were freed, our `Q2` free-list will look like:



If our evil allocation will be performed in this state, then we will get block E at address `0x1400` and overflow bytes past the address `0x1500`, and we don't know what's there. Instead, we would like to overflow a free raw buffer from our bucket, so that we will overflow the next pointer in the block. For this reason, we need to change the order of the first two allocations, at least. Reversing the order of two free blocks can be done by causing two blocks to be allocated and then freed in the same order. In our hypotetical scenario, block E and D and would be allocated, then block E and D would be freed, in the same order. The result of this operation is



Now, our evil allocation will fall in block D and we can overflow a free raw buffer, just like we wanted.

This is a nice theoretical experiment. A primitive of this kind still remains to be found.

3.6 The Allocation Primitive

At first we tried to cause an allocation of a specific size by sending IP packets with a corresponding payload size. It turns out that, at least in Digi, packets don't fall in `Q2` regardless of their payload size. In theory, with some shaping we could overflow a block from a different bucket but this would be more complex than needed.

We found that we can achieve allocation for the 0x100 bucket if an ICMP error packet is generated by the network stack. When a new ICMP error packet is generated (by the function `tfIcmpErrPacket`), a call to `tfGetSharedBuffer` is used to create the new error packet:

Ripple20 - © 2020 JSOF Ltd.

```
packetPtr = tfGetSharedBuffer(0x24,0x4c,0);
```

This causes an allocation in the correct bucket. The exact calculation is omitted.

One way to generate these error packets in a controllable way is to send a single fragment of a multi-fragment packet (a fragment with `MF=1`), but never send the closing fragment (one with the same *identification* and `MF=0`), leaving the packet "half-open", waiting to be abandoned. This causes the network stack to start a reassembly timer. When this timer times-out and no closing fragment arrives, an ICMP error packet will be generated with error type "Time Exceeded" (11) and error code "Fragment reassembly time exceeded" (1).

We can use this primitive to shape the 0x100 bucket list in the following manner:

- Send a few "half-open" fragments, each with a separate *identification* value.
- Wait for the reassembly timer to time-out on all of them. The first ICMP error sent is also the first one to be freed.

- Send two half-open fragments. This causes the order of the first two raw buffers to be reversed.
- Wait for the reassembly timer to time-out.

This primitive works, but has some drawbacks:

- By default, the maximum size of the fragmented packet queue is 5. This means that we can only achieve 5 allocations in the `Q2` bucket. This number is fairly small, but sufficient.
- The initial value for the reassembly timer is 64 seconds, by default. This means that we need to wait 64 seconds before our allocations take place. This can be a concern, because in the 64 seconds window, packets can be sent to/from the device, potentially destroying our shape. Fortunately, we found that 0x100 allocations aren't so common, so we are good to go.

3.7 Overwriting the Stack

Previously, we described how to shape the heap layout so that both the overflowing raw buffer is of known size (0x100) as well as the free raw buffer which lies directly after it. This means we can deterministicly overwrite the next pointer of the free-list with a pointer on the stack.

In the previous section, we established that the ICMP error packet primitive is good for shaping the heap. Is it also good for overwriting the stack with data we control? At first sight it seems that the answer is negative - an ICMP error packet is generated by the network stack to notify an offending host that something went wrong. However, according to [3], the ICMP error packet also includes the IP header of the offending packet along with an additional 8 bytes of payload. This information is included in order to let the offending host know what went wrong.

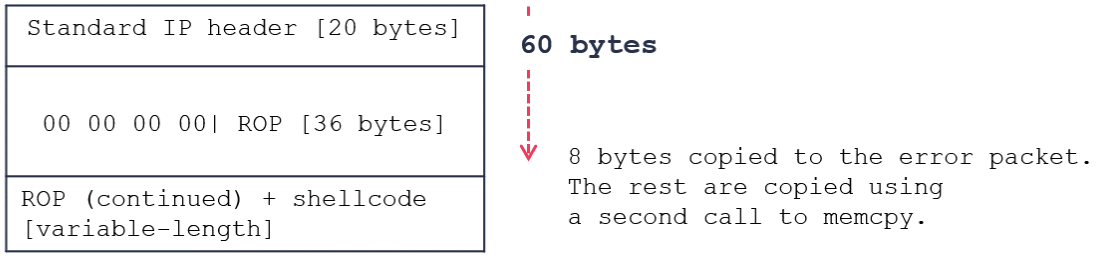
This allows us to perform a write-what-where, which we will use to write on the stack: we control the IP payload data, so we can overwrite a location on the stack with 8 bytes. However, this is insufficient if we want to write meaningful ROP and shellcode. We can improve this primitive by (ab)using the IP options that are part of the IP header included in the ICMP error packet.

Recall that the maximum IP header length is 60 bytes (`IHL=0xf`). This leaves us with an options field of 60-20=40 bytes. The options field has a specific format, so if we simply overload it with arbitrary binary data, the network stack will reject the packet prematurely. Fortunately,

Ripple20 - © 2020 JSOF Ltd.

we can mark the end of the options field using a null byte right at the beginning of the field. This causes the network stack to stop parsing the options field, bypassing rejection but still including the arbitrary data in the ICMP error message.

To align our writes to 4-byte boundary, we consumed 4 bytes from the options field and filled them with null bytes. The rest of the options field (36 bytes) contains arbitrary binary data. Accounting also for the 8 bytes copied from the IP payload, we can overwrite a location on the stack with 44 bytes of binary data. Not bad. We'll use this space for initial ROP and shellcode.



The 44 bytes overwrite will not be sufficient for all the ROP and shellcode we will end up needing, and we want to extend it. We can do this by calling `memcpy` to copy an additional section . of ROP/shellcode directly after our existing data
Also, we still need to find a suitable location on the stack to overwrite. We can determine the . stack address in advance since there is no ASLR implemented
We chose to overwrite the stack frame of `memcpy` (called from `tfIcmpErrPacket`). A nice artifact of overwriting the stack frame of `memcpy` is that we can return directly to `memcpy` a second time, extending our payload. In our specific firmware, `memcpy` keeps track of the source buffer pointer using the `r1` register which is also used as the `src` argument when `memcpy` is called. This means that when `memcpy` returns, `r1` points exactly to the location from which we stopped .copying and we thus can extend our usable stack overwrite, by back ROPing into `memcpy`

The `memcpy` function returns with the following instruction:

```
ldmia      sp!, {r0 r4 r5 r6 r7 r8 r9 r10 r11 pc}
```

Since `memcpy` expects 3 arguments, `dst`, `src` and `length`, in registers `r0`, `r1` and `r2`, respectively, we need some gadget that will allow us to set the arguments correctly (specifically, `r2`).

We used the following gadget (`gadget1`):

```
mov r2, r5; mov lr, pc; bx r7;
```

and overwrote the stack frame of `memcpy` with:

In summary, using a large IP payload, we can write a long ROP chain and shellcode, without restrictions on the data written.

3.8 Executing Shellcode

Our ultimate goal is to run shellcode. In the previous section we dropped the shellcode onto the stack. Since an XN mitigation is not present in our case, we can simply jump to the stack. 90's style.

Ripple20 - © 2020 JSOF Ltd.

```
dest address          # r0 (original stack address + 44)
0                     # r4
length to copy        # r5
0                     # r6
memory address        # r7 (address after function prologue)
0                     # r8
0                     # r9
0                     # r10
0                     # r11
gadget1 address       # pc
```

#

There's a little caveat when jumping to the shellcode in ARM - cache coherence [4]. ARM maintains separate data and instruction caches. The result is that when we overwrite the stack, the shellcode is actually written to the data cache, not to memory. Later when we jump to our shellcode, the CPU will fetch the instructions directly from memory - executing old data on the . stack
We need to clear the caches somehow. The easiest way to do it is to call a sleep function (in our case `tx thread sleep`) as part of the ROP chain, which will trigger a context switch. After .executing the sleep function, we jump to our shellcode

Gadget	Code	Purpose
gadget1	`0x0002f95c: mov r2, r5; mov lr, pc; bx r7;`	call memcpy to extend our shellcode
gadget6	`0x000267e4: pop {r0-r8, sb, sl, fp, ip, lr}; mov pc, lr;`	setup registers for next gadget
gadget5	`0x000d1c84: mov lr, pc; bx r3; mov r0, r4; pop {r4, pc};`	call sleep. return to shellcode

Figure 3.5: Full ROP chain

The full ROP chain is available in figure 3.5. The shellcode itself is nothing fancy. We made a LED blink indefinitely as a proof-of-concept.

3.9 Putting It All Together

To recollect, the exploit consists of three main stages:

1. Shaping: at this stage we send 32 ICMP echo request packets so that enough `tsPacket` structures will be allocated in the recycle list. Also, we send 5 "half-open" fragments, waiting for the reassembly timer to time-out and then send an additional 2 "half-open" fragments to reverse the order of the first two allocations.

2. Overflow: at this stage we overflow a free raw buffer (heap block) from the 0x100 bucket, so that the next pointer of the free-list points to the stack frame of `memcpy`. UDP packets are destined to port 2362 (digiman).

3. Profit: this stage causes three raw buffers of size 0x100 to be allocated, using the ICMP error packet primitive. The last of them is allocated on the stack frame of `memcpy`, and contains the ROP chain as well as the shellcode. This is the point where the CPU execution is controlled.

Ripple20 - © 2020 JSOF Ltd.

Chapter 4

CVE-2020-11898 Overview

As we've already established in 2, the Treck TCP/IP stack does not properly handles incoming IPv4 fragments over an IP-in-IP tunnel. This could also allow an unauthenticated attacker to leak memory from the heap.

Recall that in 3.7 we abused the creation of ICMP error packets to achieve write-what-where primitive. We were able to do it since the IP header of the offending packet plus 8 bytes of IP payload are copied into the error packet. This behavior can be exploited into an information leakage vulnerability if `tfIcmpErrPacket` copies out-of-bound data into the error packet.

Consider the following example:

- Inner IP packet: `IPv4{ihl=0xf, len=100, proto=0}` with payload `'\x00'*40+'\x41'*100`.
- Outer IP packet (fragment 1): `IPv4{frag_offset=0, MF=1, proto=4, id=0xabcd}` with 24 bytes from the inner IP packet payload. This means that 20 bytes of IP header will be copied, plus 4 null bytes.
- Outer IP packet (fragment 2): `IPv4{frag_offset=24, MF=0, proto=4, id=0xabcd}` with the rest of the bytes from the inner IP packet as payload.

When the network stack receives the two fragments, it reassembles them using `tfIpReassemblePacket`. This function links the two fragments using the field `pktuLinkNextPtr` in `tsUserPacket` structure (see 2.3). If tunneling is enabled, the inner IP packet will be processed next by the IP layer, in the function `tfIpIncomingPacket`.

The inner IP packet passes the IP header sanity checks since only the `pktuChainDataLength` field of `tsUserPacket` is taken into account (rather than `pktuLinkDataLength`). Also, the IP options parsing passes since there are 4 null bytes after the standard IP header (20 bytes), and a null byte represent the end of options list (see [1]).

If the total length field from the IP header of the inner IP packet is strictly less than the chain-data-length, the network stack will attempt to trim the packet. As described in 2.3, trimming is obtained by setting both the fields `pktuLinkDataLength` and `pktuChainDataLength` to the same value, the total length field (100 in our example).

Since the inner IP packet contains an invalid IPv4 protocol number (protocol 0), the network stack will reject the packet by sending an ICMP error message with type 3 (destination unreachable) code 2 (protocol unreachable). The function responsible for creating the error packet is `tfIcmpErrPacket`. It allocates a new packet, initializes some ICMP fields, and eventually copy some data from the offending packet (the inner IP packet).

Let's look more closely at the copy part:

```
1 length = (packetPtr->pktUserStruct).pktuLinkDataLength;
2 if (headerLengthInBytes + 8 <= length) {
3     length = headerLengthInBytes + 8;
```


Ripple20 - © 2020 JSOF Ltd.

```
4  }
5  memcpy(icmpErrHdrPtr + 8, pktIpHdrPtr, length);
```

As we can see, `tfIcmpErrPacket` computes the number of bytes to be copied by taking the minimum between the IP header length in bytes plus 8 (in our case, $60 + 8 = 68$) and the `pktuLinkDataLength` field, which was trimmed to be 100 in this case. Since the *actual* link-data-length of the first fragment of the offending packet is 24 (not 100), `tfIcmpErrPacket` will copy $68 - 24 = 44$ bytes of data leaked from the heap.

This vulnerability can be used to exploit CVE-2020-11896 and other RCE vulnerabilities when exploit mitigations (like ASLR) are enabled, as well as in cases where a debugger is not present.

Ripple20 - © 2020 JSOF Ltd.

Chapter 5

References

[1] *RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification*. Internet Engineering Task Force, September 1981. URL http://tools.ietf.org/html/rfc791.

[2] C. Perkins. IP Encapsulation within IP. RFC 2003 (Proposed Standard), October 1996. URL http://www.ietf.org/rfc/rfc2003.txt.

[3] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. URL http://www.ietf.org/rfc/rfc792.txt. Updated by RFCs 950, 4884.

[4] Senrio. Why is my perfectly good shellcode not working?: Cache coherency on mips and arm, 2019. URL https://blog.senr.io/blog/why-is-my-perfectly-good-shellcode-not-working-cache-coherency-on-mips-and-arm.

[5] W. Simpson. IP in IP Tunneling. RFC 1853 (Informational), October 1995. URL http://www.ietf.org/rfc/rfc1853.txt.

Ripple20 - © 2020 JSOF Ltd.

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.