ARMIS

URGENT/11

Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS

Ben Seri
Gregory Vishnepolsky
Dor Zusman

Introduction

Armis researchers discovered 11 zero day vulnerabilities in VxWorks, the most popular real-time operating system (RTOS), used by over 2 billion devices including mission-critical devices, such as industrial, medical and enterprise devices. Dubbed 'URGENT/11', the vulnerabilities reside in IPnet, VxWorks' TCP/IP stack, impacting versions for the last 13 years, and are a rare example of vulnerabilities found to affect the operating system. In its 32-year history, only 13 CVEs have been listed by MITRE as affecting VxWorks, none of which affected the core networking stack as severely as URGENT/11 does.

Vulnerabilities in widely used implementations of TCP/IP stacks have become extremely rare in recent years, especially those that can enable remote code execution on target devices. This type of vulnerabilities represent the holy grail for attackers, since they do not depend on the specific application built on top of the vulnerable stack and only require the attacker to have network access to the target device, which makes them remotely exploitable by nature. When such vulnerabilities are found in TCP implementations, they can even be used to bypass Firewall and NAT solutions as they hide within innocent-looking TCP traffic.

The 11 vulnerabilities found are comprised of 6 critical vulnerabilities, that can lead to remote code execution:

1. Stack overflow in the parsing of IPv4 packets IP options (CVE-2019-12256)
2. TCP Urgent Pointer = 0 leads to integer underflow (CVE-2019-12255)
3. TCP Urgent Pointer state confusion caused by malformed TCP AO option (CVE-2019-12260)
4. TCP Urgent Pointer state confusion during connect to a remote host (CVE-2019-12261)
5. TCP Urgent Pointer state confusion due to race condition (CVE-2019-12263)
6. Heap overflow in DHCP Offer/ACK parsing in ipdhcpc (CVE-2019-12257)

And 5 vulnerabilities that can lead to denial-of-service, logical errors or information leaks:

1. TCP connection DoS via malformed TCP options (CVE-2019-12258)
2. Handling of unsolicited Reverse ARP replies (Logical Flaw) (CVE-2019-12262)
3. Logical flaw in IPv4 assignment by the ipdhcpc DHCP client (CVE-2019-12264)
4. DoS via NULL dereference in IGMP parsing (CVE-2019-12259)
5. IGMP Information leak via IGMPv3 specific membership report (CVE-2019-12265)

This document will detail the various esoteric and somewhat forgotten mechanisms of TCP/IP that have been found to contain vulnerabilities in VxWorks' network stack implementation, as well as the vulnerabilities themselves. The whitepaper will also demonstrate the severe consequences these vulnerabilities have, affecting an extremely wide range of devices.

For more information on URGENT/11 please visit https://armis.com/urgent11

Who we are

Armis Labs is Armis' research team, focused on mixing and splitting the atoms that comprise the IoT devices that surround us - be it a smart personal assistant, a benign looking printer, a SCADA controller or a life-supporting device such as a hospital bedside patient monitor.

Our previous research includes:

- BlueBorne - An attack vector targeting devices via RCE vulnerabilities in Bluetooth stacks used by over 5.3 Billion devices. This research was comprised of 3 technical whitepapers:
    - *BlueBorne - The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks*
    - *BlueBorne on Android - Exploiting an RCE Over the Air*
    - *Exploiting BlueBorne in Linux-Based IoT deices*
- BLEEDINGBIT - Two chip-level vulnerabilities in Texas Instruments BLE chips, embedded in Enterprise-grade Access Points. The technical whitepaper for this research can be found here:
    - *BLEEDINGBIT - The hidden attack surface within BLE chips*

Why research RTOS network stacks?

In terms of raw numbers, the amount of embedded Microcontrollers and CPUs that run an RTOS is far greater, as far as we can tell, than CPUs or devices running fully-fledged OSs.

In today's world, those embedded products and components are becoming increasingly more connected to LANs and even directly to the Internet. Moreover, critical devices that comprise our infrastructure, are likely to have at least some components that use an RTOS. Therefore the impact of serious vulnerabilities in popular RTOSs is great and not well understood to date.

On top of all that, the codebases of those RTOSs are usually closed sourced, and in most cases, receive little security research into them.

Executive summary

URGENT/11 is a set of 11 vulnerabilities found to affect IPnet, VxWorks' TCP/IP stack. Six of the vulnerabilities are classified as critical and enable Remote Code Execution (RCE). The remaining vulnerabilities are classified as denial of service, information leaks or logical flaws. As each vulnerability affects a different part of the network stack, it impacts a different set of VxWorks versions. As a group, URGENT/11 affect VxWorks' versions 6.5 and above with at least one RCE vulnerability affecting each version. The wide range of affected versions spanning over the last 13 years is a rare occurrence in the cyber arena and is the result of VxWorks' relative obscurity in the research community. This timespan might be even longer, as according to Wind River, three of the vulnerabilities have already existed in IPnet when it was acquired from Interpeak in 2006.

URGENT/11 enables three attack scenarios, depending on the location of the device on the network and the attacker's position.

The first attack scenario affects any impacted VxWorks device stationed at the perimeter of the network and is directly exposed to the Internet such as firewalls, modems, routers. An attacker can directly attack such devices from the Internet, compromise them and subsequently compromise the networks they guard.

The second attack scenario affects any impacted VxWorks device stationed behind the perimeter, inside an internal network, that connects outbound to the internet through Firewall or NAT solutions. URGENT/11 can allow an attacker to take over such devices, by intercepting TCP connections they create to the Internet, and manipulating certain field of the TCP header in packets that are sent back through Firewall or NAT solutions. This is due to the vulnerabilities' uniquely low level position inside the parsing and handling of the TCP header.

In the last scenario, an attacker already positioned within the network as a result of a prior attack can send a specially crafted broadcast IP packet that will hit all vulnerable VxWorks devices within the local LAN at once. This is due to a very unique vulnerability found in the parsing and handling the IP header, that is triggered even in broadcast. This vulnerability is also an RCE vulnerability that can lead to remote take over.

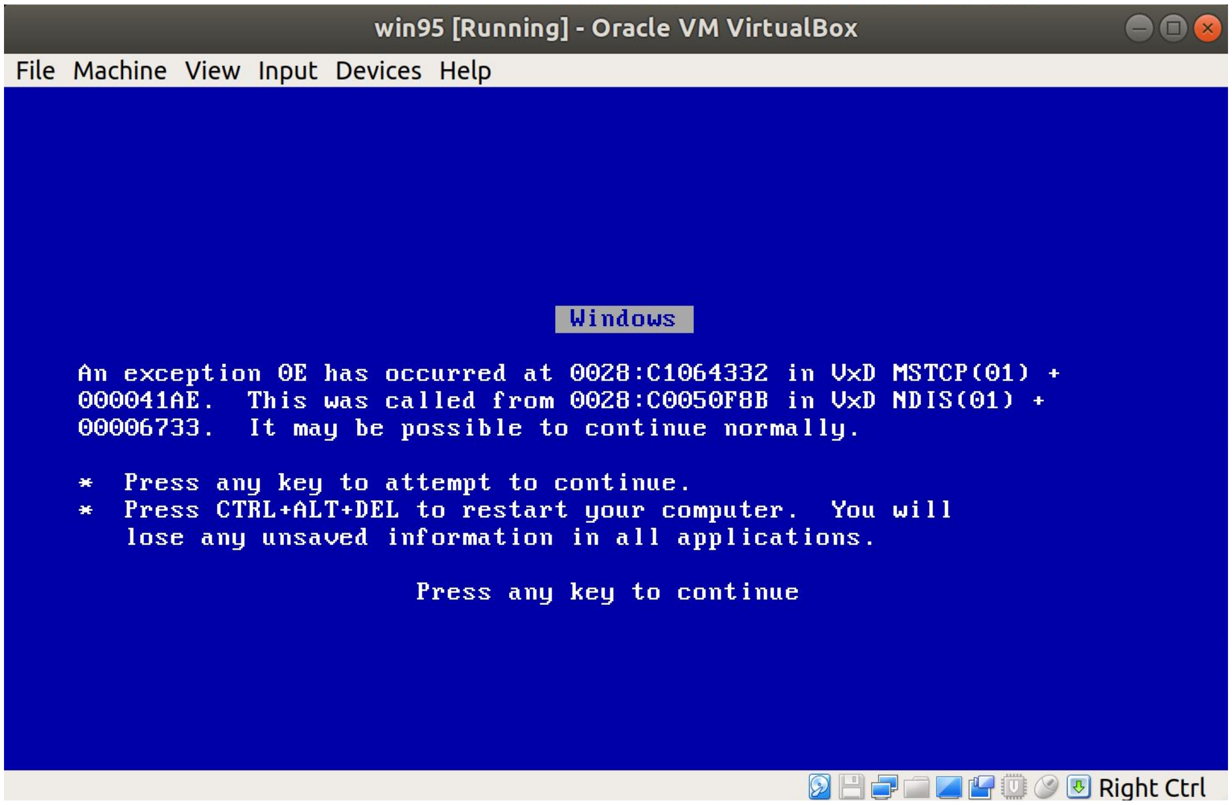
Preparing the groundwork

Past research

Implementation vulnerabilities in various layers of the TCP/IP stack were relatively common in the past. Unsurprisingly, 1990's software and operating systems had various DoS and RCE bugs in their network stacks. At the time, Internet connected PCs were only beginning to become popular, so even trivial bugs (by today's standards) were to be expected. However, over the years, network stacks came under justified scrutiny, and today such vulnerabilities are an extreme rarity.

In this research we've looked into a more modern and up-to-date implementation of the TCP/IP stack. Interestingly enough, the bugs we found in our modern RTOS suffered from the same pitfalls as their ancient counterparts.

For example, the WinNuke bug that became known in 1997, and was subsequently widely exploited, was a remote DoS bug in Windows 95 and NT. The bug could be triggered simply by sending a single TCP out-of-band segment (with the URG flag set) to a windows machine, resulting in a BSOD.


win95 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
                          Windows

An exception 0E has occurred at 0028:C1064332 in VxD MSTCP(01) +
000041AE.  This was called from 0028:C0050F8B in VxD NDIS(01) +
00006733.  It may be possible to continue normally.

*  Press any key to attempt to continue.
*  Press CTRL+ALT+DEL to restart your computer.  You will
   lose any unsaved information in all applications.

                      Press any key to continue
```

Actual reproduction of the WinNuke bug on Windows 95 running inside VirtualBox. The crash is in the MSTCP VxD driver, which implements the TCP protocol.

This is remarkably similar to CVE-2019-12255 (“TCP Urgent Pointer = 0 leads to integer underflow”) which is presented in this paper. In our case, however, the bug is classified as an RCE (memory corruption).

An additional example of ancient TCP/IP implementation bugs was the Ping Of Death, a DoS attack that affected many different operating systems back in the day - including Windows, Linux, Mac and Unix. This simple attack was generated by sending an ICMP Echo request (ping) with a payload of 64KB of data (the maximum possible payload in an IP fragmented packet). Back in 1997, this was all it took to remotely crash a wide array of operating systems.

Surprisingly, modern examples of remotely triggerable TCP/IP bugs can still be found in the most widely used OSs, albeit being an extremely rare occurrence. For instance, CVE-2019-0547 is an RCE in the Windows 10 DHCP client. This bug can be triggered by malformed DHCP options, much like CVE-2019-12257 (“Heap overflow in DHCP Offer/ACK parsing inside ipdhcpc”) that is presented in this document.

Another example is Apple’s heap overflow bug from 2018 affecting all Apple devices and OSs, namely CVE-2018-4407 [1][2][3]. This one is a bug in the construction of an ICMP error packet that is sent in response to a malformed IPv4 packet with specially crafted IP options. This too is remarkably similar to our CVE-2019-12256 (“Stack overflow in the parsing of IPv4 packets IP options”), which in our case is an RCE.

Recent research into the TCP/IP implementation of AWS FreeRTOS (an RTOS by Amazon for IoT devices) has also led to the discovery of many RCE bugs (CVE-2018-16522 through CVE-2018-16528).

Researching VxWorks

VxWorks is not an open source product. Wind River, the company that originally authored VxWorks, is still updating it and selling it today. When their customer buys a license to embed the VxWorks RTOS into their product, what they generally receive is the Wind River Workbench IDE. This IDE comes with the VxWorks source code, and a wide variety of hardware support packages (BSPs).

In the past, it was possible to obtain a working BSP image for VMWare directly from their website, as is described in this post. Today, it’s still possible to obtain a similar evaluation product after contacting their sales team, however this approach is likely not useful for security researchers.

While legally obtaining the source code for research purposes is difficult, binary analysis is effective in the case of VxWorks research. There is a multitude of real world products that run up to date versions of VxWorks available for purchase, and their firmwares are usually freely available for download online. Due to the nature of how the Wind River Workbench IDE produces binaries, these firmwares are commonly shipped with ELF files containing full debug symbols, making them easy to decompile with modern tools, arriving at high quality results.

Some of the aforementioned products also come equipped with internal UART and JTAG interfaces, making debugging easy for a determined researcher. These qualities result in a setup that is as useful for research purposes as the BSP image mentioned above. In case the researcher has access to *any* version of the IPnet source code (from version 6.5 and above), the firmware analysis can be complemented to a degree where the resulting material is of source code quality for any desired VxWorks version.

Identifying the weak points in TCP/IP implementations

TCP/IP vulnerabilities from the past, across many different OSs and software implementations, have common weak points. Certain areas in the specifications have an inordinate amount of implementation bugs. In some cases, the same exact bugs appear across completely independent implementations. When approaching the task of discovering security issues in the VxWorks TCP/IP stack, we took the following map of weak points as a guide. Most of the bugs we've found were rather non-trivial, unlike their counterparts from the 90's. However, all of them were somewhat related to the following parts of TCP/IP:

- Parsing and handling of IPv4 options
    - Structure of these options can become quite complex.
    - Many IP options are considered highly esoteric, and are rarely used in practice, but are still be parsed by the various stacks.
- Parsing and handling of TCP options
    - Similarly to the IP options, some TCP options no longer make sense on the modern Internet, but most are still implemented in various TCP stacks.
    - For instance, the *MD5 Signature Option* and its rather modern successor *TCP Authentication Option (TCP-AO)* have pretty much no place in a world where TLS is king. Consequently, they are almost never used (or tested).
- ICMP error packets
    - Various error conditions in the handling of IPv4 packets, as well as the higher level TCP, UDP and ICMP protocols, may result in the sending of ICMP error packets.
    - The structure of these error packets is non-trivial, as they sometimes contain a copy (sometimes modified) of the original bad packets. Additionally, certain IP options affect their handling as well.
- TCP Urgent / out-of-band data
    - Per the RFC, TCP supports sending and receiving out-of-band data, using the URG flag and the Urgent Pointer field that exists in **every** TCP segment.
    - However, its exact behaviour is very poorly defined, and consequently extremely badly implemented (more on this later).
- IP datagram fragmentation
    - IPv4 (and v6) support fragmentation of datagrams at the IP layer. A 64kb packet can be fragmented according to the underlying MTU.
    - In practice, this feature is rarely used, thus the combination of fragmented IP packets, with various protocols on top of IP can introduce interesting edge cases.

Six critical RCE vulnerabilities discovered

As mentioned above, six of the discovered vulnerabilities are of critical severity, since they are memory corruptions that can lead to remote code execution. The six vulnerabilities were found in three separate subsystems of VxWorks' TCP/IP stack (IPnet):

1. One RCE vulnerability in the IP layer (CVE-2019-12256)
2. Four RCE vulnerabilities in the TCP layer (CVE-2019-12255, CVE-2019-12260, CVE-2019-12261 & CVE-2019-12263)
3. One RCE vulnerability in the IPnet's built-in DHCP client, ipdhcpc (CVE-2019-12257)

The following sections will provide some background on the mechanisms in which these vulnerabilities were found, and detail the vulnerabilities themselves.

Stack overflow in the parsing of IPv4 packets' IP options

TL;DR

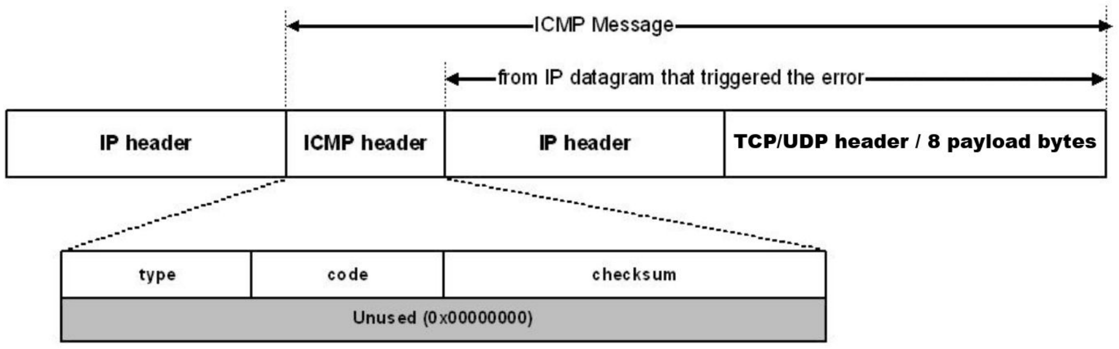
When sending a malformed IP packet containing multiple Source Record Route (SRR) options to a vulnerable VxWorks device, an ICMP error response packet is generated in response. The SRR options are copied into the IP options of the response packet without proper length validations, which result in an attacker controlled stack overflow.

Background - ICMP error packets, IP options and Loose Source Routing

ICMP error packets

ICMP error packets are sent as a response to an error condition that arises during the handling of an IP packet. They are usually addressed to the source IP address of the packet. While most ICMP errors indicate network problems, like routing loops or an inability to route to the destination IP, some arise due to problems within the incoming (bad) packets themselves. For example, the *Parameter Problem* error is sent in response to an IP packet with malformed IP options.

Many of these error packets will also contain a copy of the incoming bad packet, that triggered the problem. This copy will contain the original IP header, the TCP or UDP header, and some additional bytes from the payload of the packet.



Structure of an ICMP error packet

Some implementations fail to reconstruct these headers into the ICMP error packet correctly, due to certain edge cases that can arise when handling malformed data.

An example of such an issue is Apple's XNU kernel heap overflow bug from 2018 (CVE-2018-4407). An integer underflow bug existed in the function *icmp_error* (bsd/netinet/ip_icmp.c:313). The bug could be triggered if the length of the bad packet's IP header + TCP header is more than 84 bytes. Recall that both the IP and TCP headers have an options field, thus both can be up to 60 bytes long each.

A TCP packet with a valid 40 bytes long TCP options field, and an invalid 40 bytes long IP options field will cause an ICMP error to be sent, and trigger the integer underflow. This, in turn, will cause a wildcopy heap overflow. An analysis of this vulnerability is available from the author.

The use of various IP options in conjunction with an ICMP error packet that is sent in response to a malformed packet can result in certain edge cases that may lead to bugs. But what is the actual functionality of the IP options themselves?

IP options, and Loose Source Routing

Each IPv4 packet has a non-mandatory options field that can be up to 40 bytes long. Each separate option appears as a TLV field (type, length, value), and all of them together must fit within the 40 bytes. An ICMP *Parameter Problem* packet will be sent if an error is encountered during the parsing of these options.

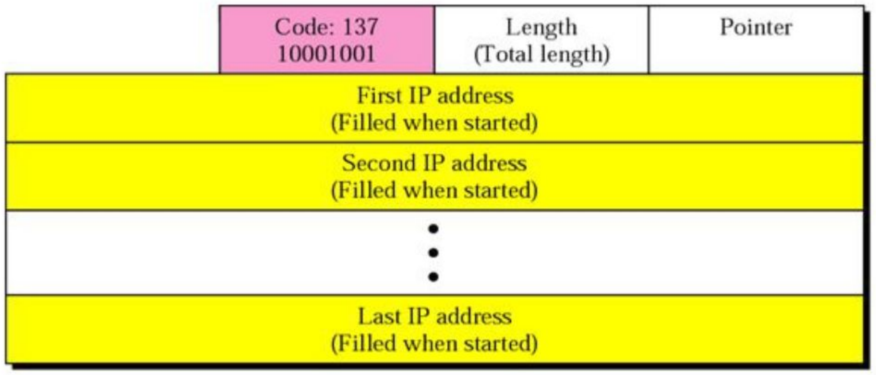
Some of the more widely known options are:

1) **Time Stamp**: Requests each router that forwards the packet to append its current timestamp into the timestamp option field. The size of the entire option is set by the sender.
2) **Record Route**: Requests each router to append its IP into the option, effectively recording the route a packet has taken.
3) **Loose Source Route (LSRR)**: Instructs the routers to route the packet via a given list of IPs, and record the route taken into the option. A longer route may be taken as long as the packet is still routed through every IP in the list.
4) **Strict Source Route (SSRR)**: Same as above, but the exact given route must be taken.
5) **Traceroute**: Requests each router to send an ICMP Traceroute packet to the source IP.

6) **Router Alert**: Alerts the routers that the packet may require more extensive processing by the routers on the way to the destination.

The infamous LSRR and SSRR options described above are known to introduce security concerns to any router that supports them, since they enable attackers to abuse the default routing path of a packet and force a malicious route on packets containing them. Due to this concern, they are blocked by almost all routers by default. However, in most TCP/IP implementations, these rather complex options are still parsed before they are blocked, presenting an additional attack surface nonetheless.

Every LSRR\SSRR option has the following structure:



The code, length and pointer fields are 1-byte wide. The IP addresses are 4 bytes each

The option contains a list of IP addresses through which the packet should be routed (either *strictly* or *loosely*). The *Pointer* is an offset within the option to the next IP address that the packet should be routed through, and it is incremented by each router through which the packet traverses.

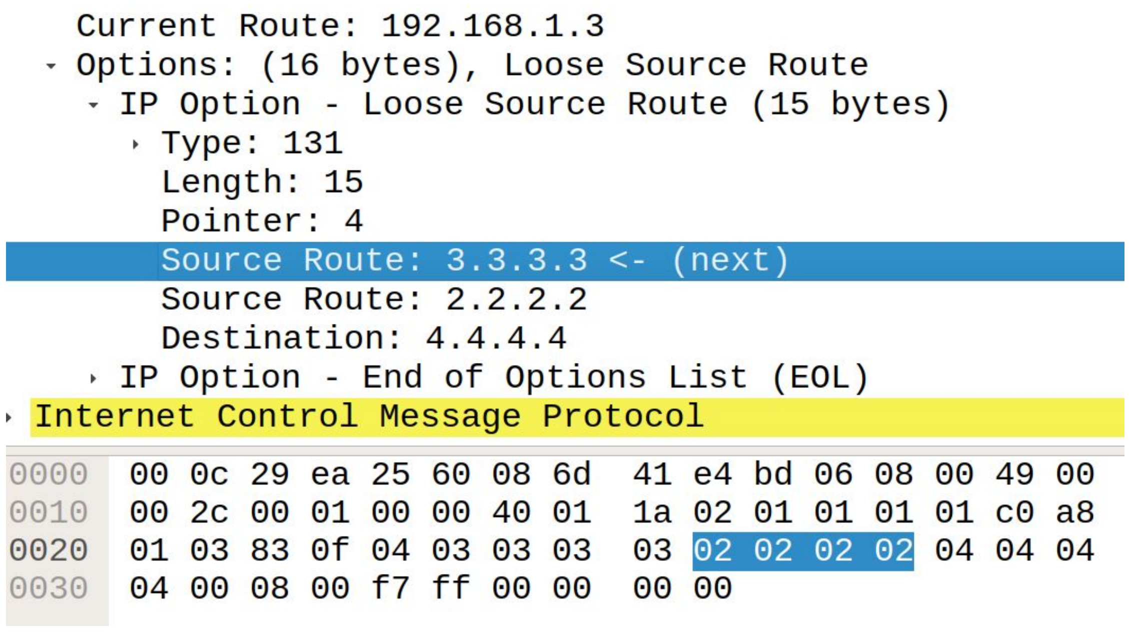
According to RFC1122, once the packet has reached its destination, the IP list in the option needs to be recorded and passed up to the transport layer or to ICMP message processing. This recorded route will be reversed and used to form a return source route for reply datagrams, or for ICMP error messages sent back to the sender.

This logic can be carried with the following copying logic:

- All the IP addresses up to, but not including, the address prior to where *Pointer* points to are copied in reverse order.
- The source IP of the packet is added to the end of the list instead of the not-included address.

The logic of both a router that supports source routing options, and of a host that supports receiving packets that contain them is complex. The Parsing process of such options is thus also subject to potential bugs.

An example LSRR option from Wireshark:



```
    Current Route: 192.168.1.3
  ▾ Options: (16 bytes), Loose Source Route
    ▾ IP Option - Loose Source Route (15 bytes)
      ▸ Type: 131
        Length: 15
        Pointer: 4
        Source Route: 3.3.3.3 <- (next)
        Source Route: 2.2.2.2
        Destination: 4.4.4.4
    ▸ IP Option - End of Options List (EOL)
▸ Internet Control Message Protocol

0000  00 0c 29 ea 25 60 08 6d  41 e4 bd 06 08 00 49 00
0010  00 2c 00 01 00 00 40 01  1a 02 01 01 01 01 c0 a8
0020  01 03 83 0f 04 03 03 03  03 02 02 02 02 04 04 04
0030  04 00 08 00 f7 ff 00 00  00 00
```

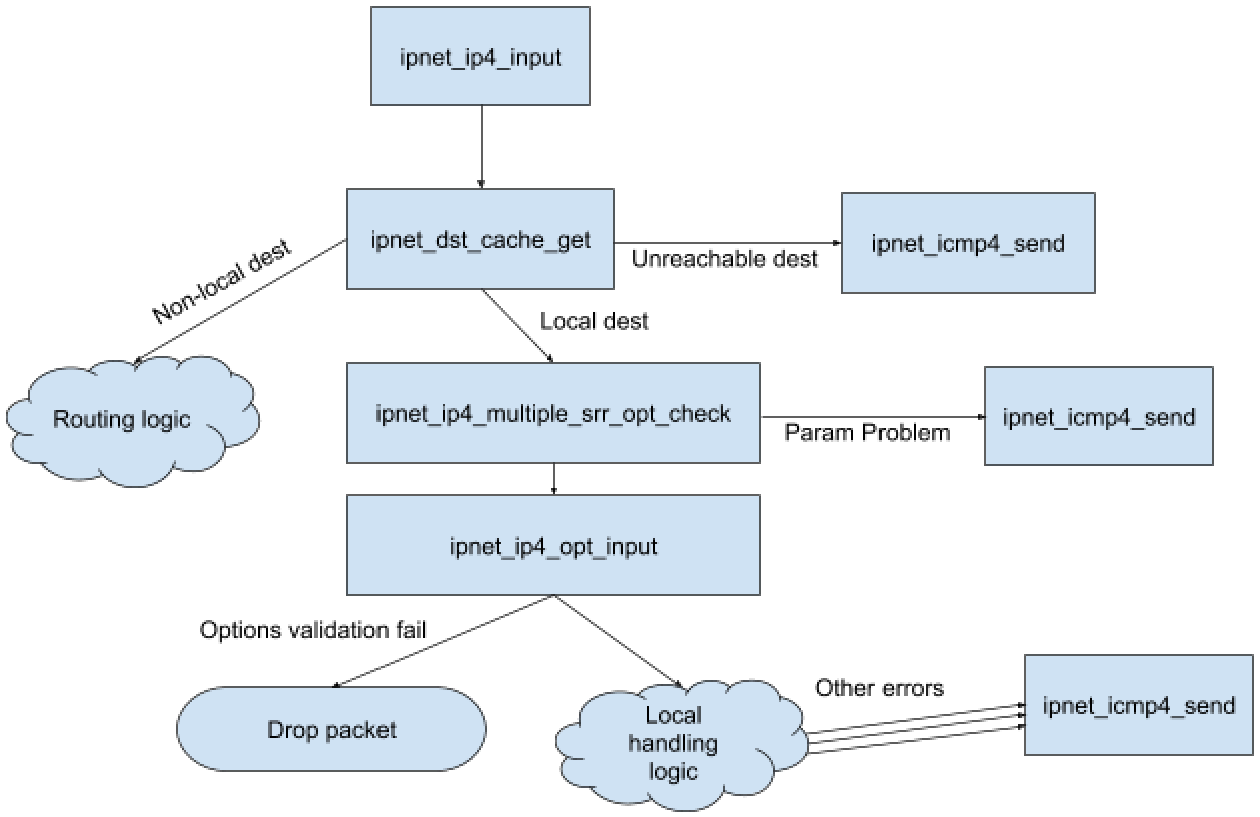
Wireshark capture of an IP packet with an LSRR option

The support for a device to act as a **host** of source routing, as described in RFC1122, is almost never implemented in practice. However, in VxWorks this support exists for some reason, introducing the attack surface for the following vulnerability.

The vulnerability (CVE-2019-12256)

In VxWorks' IPnet stack the IPv4 protocol is implemented in the module *ipnet2/src/ipnet_ip4.c*. The function *ipnet_ip4_input* is the entry point for incoming IPv4 packets. This function first performs basic validation of the header, and then calls *ipnet_dst_cache_get*. This performs a lookup for a handler callback, according to the source and destination IP addresses of the packet. For instance, if the destination address matches the local address, the *ipnet_ip4_local_rx* handler will be called. Otherwise the packet may be forwarded (routed), or perhaps a *Destination Unreachable* condition may be triggered.

Certain error conditions, such as *Destination Unreachable* will trigger an ICMP error packet to be sent by calling *ipnet_icmp4_send*. This function will also be called when an error occurs due to malformed IP options, and an ICMP *Parameter Problem* packet will be sent. This can occur through multiple stages in which the IP options are being parsed: in *ipnet_ip4_multiple_srr_opt_check,* which validates that only one instance of either LSRR or SSRR options exists in the option field, or in any of the various option parsing functions - *ipnet_ip4_opt_*_rx*. Inside *ipnet_icmp4_send*, an error packet is constructed, and certain options fields from the incoming (malformed) packet are copied. This is done by the *ipnet_icmp4_copyopts* function. The above flows are illustrated below:



IPv4 packet handling flow chart, with calls to the ICMP error sending function

As shown above, while parsing incoming IPv4 packets, various code flows can lead to ICMP messages being sent in response to erroneous (malformed) packets. The *ipnet_icmp4_send* function will be used to send the response ICMP packets, and it will attempt to copy certain IP options from the incoming packet onto the outgoing packet with the function *ipnet_icmp4_copyopts*. In at least two code flows, the outgoing ICMP packet will be sent **before** the incoming packet is fully parsed, and the incoming IP options are fully validated to be legal, or even despite them failing validation already. This design flaw can lead to a stack overflow in the context of this *ipnet_icmp4_send*.

This vulnerability exists since VxWorks version 6.9.3.

Two distinct flows can lead to this vulnerability:

1. A packet that is sent to the target's MAC address, but with a destination IP address that is not a unicast address of the target, and is unreachable for it, which would result in a call to *ipnet_ip4_dst_unreachable*.
2. A normal unicast packet that is directed to the target with both its MAC and IP addresses. This will occur in the *ipnet_ip4_multiple_srr_opt_check* function.

The first flow happens in *ipnet_ip4_input*, when the destination IP of the incoming packet is unreachable according to *ipnet_dst_cache_new's* return value, which indicates how the packet should be routed:

```
    return_code = ipnet_dst_cache_new(&v37, ipnet_ip4_dst_cache_rx_ctor, &v45);
    if ( return_code < 0 )
    {
      return_code = -1 * return_code;
      // If dst is unreachable (EHOSTUNREACH, ENETUNREACH or EACCES)
      if ( return_code == 51 || return_code == 65 || return_code == 13 )
      {
        ipnet_ip4_dst_unreachable(packet, return_code);
```

Decompiled snippet from *ipnet_ip4_input*

The function *ipnet_ip4_dst_unreachable* will then call *ipnet_icmp4_send* and attach the original packet to the *icmp_param* struct. In *ipnet_icmp4_send*, the IP options from the failing (incoming) packet will be referenced by a structure that will be passed to *ipnet_icmp4_copyopts*:

```
struct Ipnet_copyopts_param
{
  void *options_ptr;
  int total_opt_size;
  int which_opts_to_copy;
};

int ipnet_icmp4_send(Ipnet_icmp_param *icmp_param, Ip_bool is_igmp)
{
  Ipnet_icmp_param *icmp_param;
  Ipcom_pkt *failing_pkt;
  struct Ipnet_copyopts_param options_to_copy;
  struct Ipnet_ip4_sock_opts opts;
  ...
  options_to_copy.options_ptr = NULL;
  options_to_copy.total_opt_size = 0;
  // By default, copy SSRR and LSRR options.
  options_to_copy.which_opts_to_copy = 0x208;
  ...
  if ( !is_igmp )
  {
    if ( icmp_param->type == IPNET_ICMP4_TYPE_DST_UNREACHABLE ||
         icmp_param->type == IPNET_ICMP4_TYPE_PARAMPROB ||
         ... ) {

      failing_pkt = icmp_param->recv_pkt;
      ...
      ip_hdr = failing_pkt->data[icmp_param->recv_pkt->ipstart];
      ...
      options_to_copy.total_opt_size = 4 * (*ip_hdr & 0xF) - 20;
      if ( options_to_copy.total_opt_size )
        options_to_copy.options_ptr = ip_hdr + 20;
      ...
    }
  }
  ...
  ipnet_icmp4_copyopts(icmp_param, &options_to_copy, &opts, &ip4_info);
  ...
}
```

Decompiled snippet from *ipnet_icmp4_send*

By default, the *options_to_copy* structure will instruct *ipnet_icmp4_copyopts* to copy the SSRR and LSRR options from the failing packet to the output packet. When the failing packet hasn't been validated to contain valid IP options (as with the above flow), the *ipnet_icmp4_copyopts* function may be coerced into copying **multiple** SSRR or LSRR options from the failing packet onto the *opts* structure that is allocated on the stack of *ipnet_icmp4_send*. Additionally, these copied options might contain illegal structures that should otherwise have not been allowed. For example, when sending an IP packet that contains the following bytes in the IP options field, a stack overflow will occur:

Type (LSRR)	Length	LSRR-Pointer	Type (LSRR)	Length	LSRR-Pointer
\x83	\x03	\x27	\x83	\x03	\x27

In this example, two LSRR option are contained in the IP options field. These LSRR options don't contain any routing entries (each option length is only 3 bytes) and the SRR-Pointer field points past the end of the option. As described previously, a host that receives an SRR option needs to take all the recorded routing entries, reverse their order, and use it to create a new SRR option that will be added to any returned packet. This functionality can be viewed in *ipnet_icmp4_copyopts*:

```
int ipnet_icmp4_copyopts(Ipnet_icmp_param *icmp_param,
                         struct Ipnet_copyopts_param *copyopts_param,
                         struct Ipnet_ip4_sock_opts *opts, void *ip4_info)
{
  ...
  while ( 1 ) {
    current_opt = ipnet_ip4_get_ip_opt_next(current_opt,
                                            copyopts_param->options_ptr,
                                            copyopts_param->total_opt_size);

    if ( !current_opt )
      break;
    opt_type = *current_opt;
    if ( copyopts_param->which_opts_to_copy & (1 << (opt_type & 31)) )
    {
      ...
      if ( opt_type == 0x83 || opt_type == 0x89 ) {
        // IP_IPOPT_LSRR or IP_IPOPT_SSRR
        srr_ptr_offset = 39;
        srr_opt = (srr_opt_t *)&opts->opts[opts->len];
        // Limits max ptr offset to 39,
        // (but doesn't validate this offset is within the current option)
        if ( (int)current_opt[2] <= 39 )
          srr_ptr_offset = current_opt[2];
        offset_to_current_route_entry = srr_ptr_offset - 5;
        ...
        srr_opt->type = opt_type;
        current_route_entry = &current_opt[offset_to_current_route_entry];
        srr_opt->length = 3;
        srr_opt->route_ptr = 4;
        while ( offset_to_current_route_entry > 0 ) {
          memcpy((char *)srr_opt + srr_opt->length, current_route_entry, 4);
          current_route_entry -= 4;
          offset_to_current_route_entry -= 4;
          srr_opt->length += 4;
        }
        memcpy((char *)srr_opt + srr_opt->length, &icmp_param->to, 4);
        srr_opt->length += 4;
        total_opts_len = opts->len + srr_opt->length;
      }
    }
  }
  ...
}
```

Decompilation output from a binary image, inside *ipnet_icmp4_copyopts*

The code in *ipnet_icmp4_copyopts* will use these SRR-Pointer fields as the offset to the final route entry in an SRR option, and copy all the routing entries up to it to the outgoing packet options *opts,* which is allocated on the stack of *ipnet_icmp4_send*. Each LSRR option in the input buffer is 3 bytes long, but it would generate a copied-out option of 43 bytes (3 bytes header, 36 bytes of routing entries, 4 bytes of a new routing entry for the current route). Since there is no validation (in this context) that the failing packet doesn't contain more than one SSRR\LSRR option, sending multiple options of this type will result in the overflow of *opts* which is a 40 bytes array allocated on stack.

As noted above, this vulnerability can also be triggered by another flow, when a packet containing the above options is sent directly to the target device, without resulting in a destination unreachable condition. This is due to the fact that when the option parsing process fails in various *ipnet_opt_*_rx* functions, or in the *ipnet_ip4_multiple_srr_opt_check* function, an ICMP error message is sent in response (via *ipnet_ip4_opt_icmp_param_prob*). When this occurs, the remaining options that **weren't** parsed by these validation functions can still contain illegal values, such as the multiple SRR options presented above, which would lead to the mentioned stack overflow. Since the overflow will contain attacker-controlled data from the input packet, this vulnerability can lead to remote code execution in both flows.

Luckily, because the vulnerability depends on sending packets with invalid IP options, it **can not** be exploited over the Internet. The first router that encounters the packet will drop it. Therefore, the vulnerability is only exploitable by an attacker on the LAN.

Since this vulnerability is in the parsing of the IP header itself, it can also be triggered by sending a specially crafted IP packet with the invalid IP options in a **broadcast** packet. This can allow an attacker to target multiple vulnerable devices **simultaneously.**

TCP Urgent Pointer RCE vulnerabilities

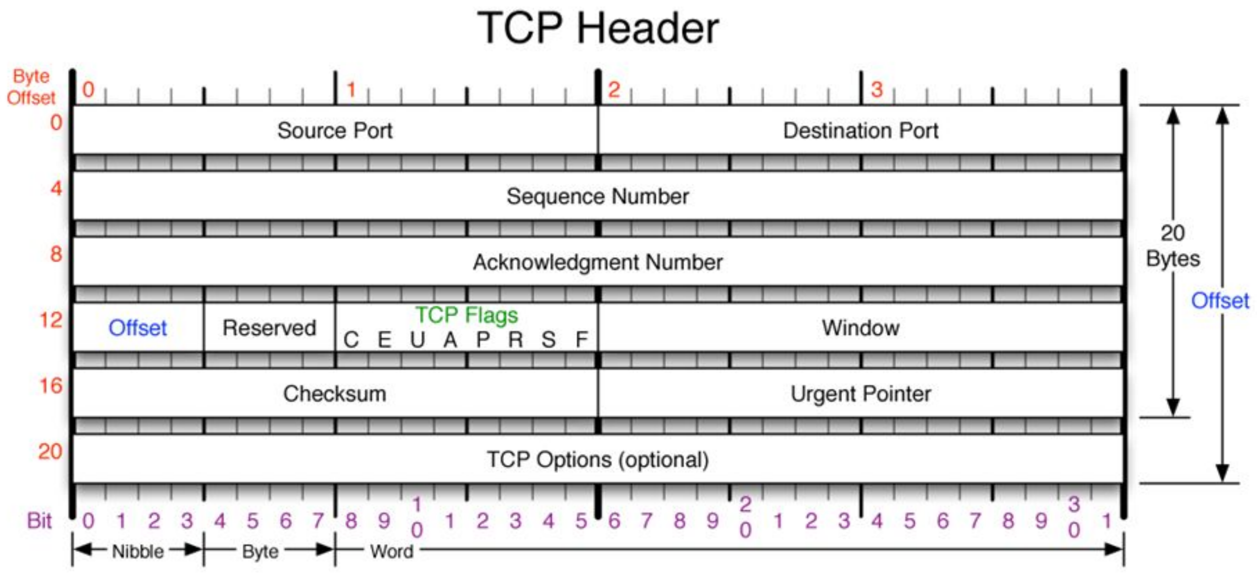
TL;DR

TCP has an esoteric mechanism to transfer out-of-bounds data named Urgent Data. By exploiting this mechanism, an attacker can underflow the length variable passed to *recv()* system calls, which will result in attacker-controlled overflow of the buffer passed to *recv()*. This can lead to overflow of a buffer allocated either in stack, heap, or global data section, which can lead to remote code execution.

Background - crash course in TCP

To understand the discovered vulnerabilities in IPnet's implementation of TCP, a quick crash course in TCP is required. TCP is a transport layer protocol that allows the transfer of an ordered stream of bytes over the unreliable IP layer. Explaining how the TCP protocol works, even at a cursory level, is well beyond the scope of this document. However, some particulars must be explained to understand the vulnerabilities described below.

Each packet sent over the IP layer as part of a TCP session is called a segment. Below is an illustration for the structure of a segment:



Each TCP segment is considered part of a particular TCP session according to its 4-tuple of source/destination IP and source/destination Port.

Each segment has a *Sequence Number* field, that allows the **receiving** endpoint to determine where the included data is supposed to appear in the stream.

- The first packet of a TCP connection that was **received** by the endpoint contains the *Initial Sequence Number (ISN)*. This segment must have the SYN flag set.
- Subtracting the ISN from all other Sequence numbers received by **that** endpoint as part of the session provides the **offset** of the included data within the stream

The TCP/IP stack of each endpoint holds a buffer called the *TCP Window*, that contains all data that was received as part of the session, but not yet handled by the user of the socket. Data is pulled out of the *TCP window buffer* when the user calls the *recv()* system call on the socket. Data is inserted whenever a new (non-duplicate) TCP segment arrives.

The ***next in-order*** *Sequence Number* is kept for the window, and data can only be *recv()'d* by the user if it's located before that offset in the stream. Therefore, the user will only ever read in-order data.

In-order data	Hole	Out-of-order data
	^ next in-order sequence number	

A new segment that arrives may complement previous out-of-order data in the window, and thus make some (or all) of this data in-order. Therefore, the *next in-order Sequence Number* will then be adjusted to reflect this, and the user may now *recv()* the data.

TCP Urgent data explained

A lesser-known feature of TCP enables sending and receiving *Urgent* or *Out of Band* data over an existing TCP connection. This feature was designed to solve problems that arise during situations similar to the following use case:

1) A client sends instructions to a server over a TCP connection asynchronously. That is, without waiting for one to be finished before sending the next.
2) The server reads an instruction from its *TCP window*, and executes it, repeating for every instruction. **However,** before all instructions were handled by the server, the client decides to cancel the remaining instructions.

With just one stream of data, it's not possible to notify the server of any new requests before all previous requests have finished processing. In modern layer 7 protocols, this issue is solved by explicitly using more than one stream of data. TCP, however, provides a method to send this *Urgent data* over the same established layer 4 connection.

Almost all modern OSs provide a standard *MSG_OOB* flag that can be passed to the *send/recv* system calls to send and receive this out-of-band (OOB) data over a TCP socket. A buffer sent as OOB data will **not** be received by a regular *recv()* call (that only receives ordered data), and a *recv()* call with the *MSG_OOB* flag set will only receive this OOB data.

At the TCP segment level, this feature is implemented by an *URG* flag and an *Urgent Pointer* field that exists in every segment. If the URG flag is set, the *Urgent Pointer* indicates the offset in the stream (from the relative *Sequence Number*) where the urgent data is located. Usually the urgent data itself will also appear in that same segment.

This sounds simple enough, however the exact meaning of the *Urgent Pointer* is not well defined - it is unclear whether it points to the last byte of the ordered data, or the first byte of the urgent data. In addition, it is unclear how one can define the length of the urgent data, having only a pointer that indicates where it starts, but not where it ends.

Multiple RFCs dating back to the early 1980s have produced contradicting answers to these questions:

RFC 793 (1981, page 17) states:

The urgent pointer points to the sequence number of the octet following the urgent data.

However, RFC 1011 (1987, page 8) states:

Page 17 is wrong. The urgent pointer points to the last octet of urgent data (not to the first octet of non-urgent data).

And RFC 1122 (1989, page 84) reinforces this approach:

..the urgent pointer points to the sequence number of the LAST octet (not LAST+1) in a sequence of urgent data.

Finally, the latest RFC to handle this issue (RFC 6093 [2011, pages 6-7]) concludes:

Considering that as long as both the TCP sender and the TCP receiver implement the same semantics for the Urgent Pointer there is no functional difference in having the Urgent Pointer point to 'the sequence number of the octet following the urgent data' vs. 'the last octet of urgent data', and that all known implementations interpret the semantics of the Urgent Pointer as pointing to 'the sequence number of the octet following the urgent data'.

So since virtually all existing TCP implementations handle the urgent pointer as the sequence number of the octet following the urgent data, this should be the default behavior of all stacks going forward.

As to the length of the urgent data, RFC 6093 (page 5) also states this:

If successive indications of 'urgent data' are received before the application reads the pending 'out-of-band' byte, that pending byte will be discarded (i.e., overwritten by the new byte of 'urgent data').

Concluding that the urgent data should always be one octet (byte).

Due to the various intricacies of the Urgent Pointer mechanism, some implementations (VxWorks included) were forced to support an *RFC-1122 compatible* mode, and a non-compatible mode - where the Urgent Data would either point ±1 of the calculated urgent pointer.

This, combined with the fact that the whole feature itself is esoteric, resulted in it being poorly implemented and tested by the various OSs. For example, the Windows NT/95 WinNuke bug described earlier in this document was triggerable simply by sending *any* OOB data to the machine, as it was *always* mis-handled by the TCP driver resulting in a blue screen of death.

TCP in the IPnet stack

In VxWorks' IPnet stack the TCP protocol is implemented in *iptcp/src/iptcp.c*. Each TCP segment received by the stack is processed by *iptcp_input*:

```
int __cdecl iptcp_input(int a1, int a2, int (__cdecl *socklookup)(...))
{
  ...
  // Lookup the TCP socket based on the incoming 4-tuple
  // (This would match the TCP listening socket, for a SYN on new 4-tuple)
  sock = socklookup(...);
  ...
  // If the packet matched a listening socket, create a new socket using it's 4-tuple
  if (sock->tcb->flags & 1)
      sock = iptcp_handle_passive_open(...);
  ...
  // Process TCP options (this might lead to CVE-2019-12260 or CVE-2019-12259)
  if ( iptcp_process_options(...) < 0 ) {

    iptcp_send_reset(...);
    return 0;
  }
  return_code = iptcp_deliver(...);
  ...
}
```

Simplified decompilation of *iptcp_input*

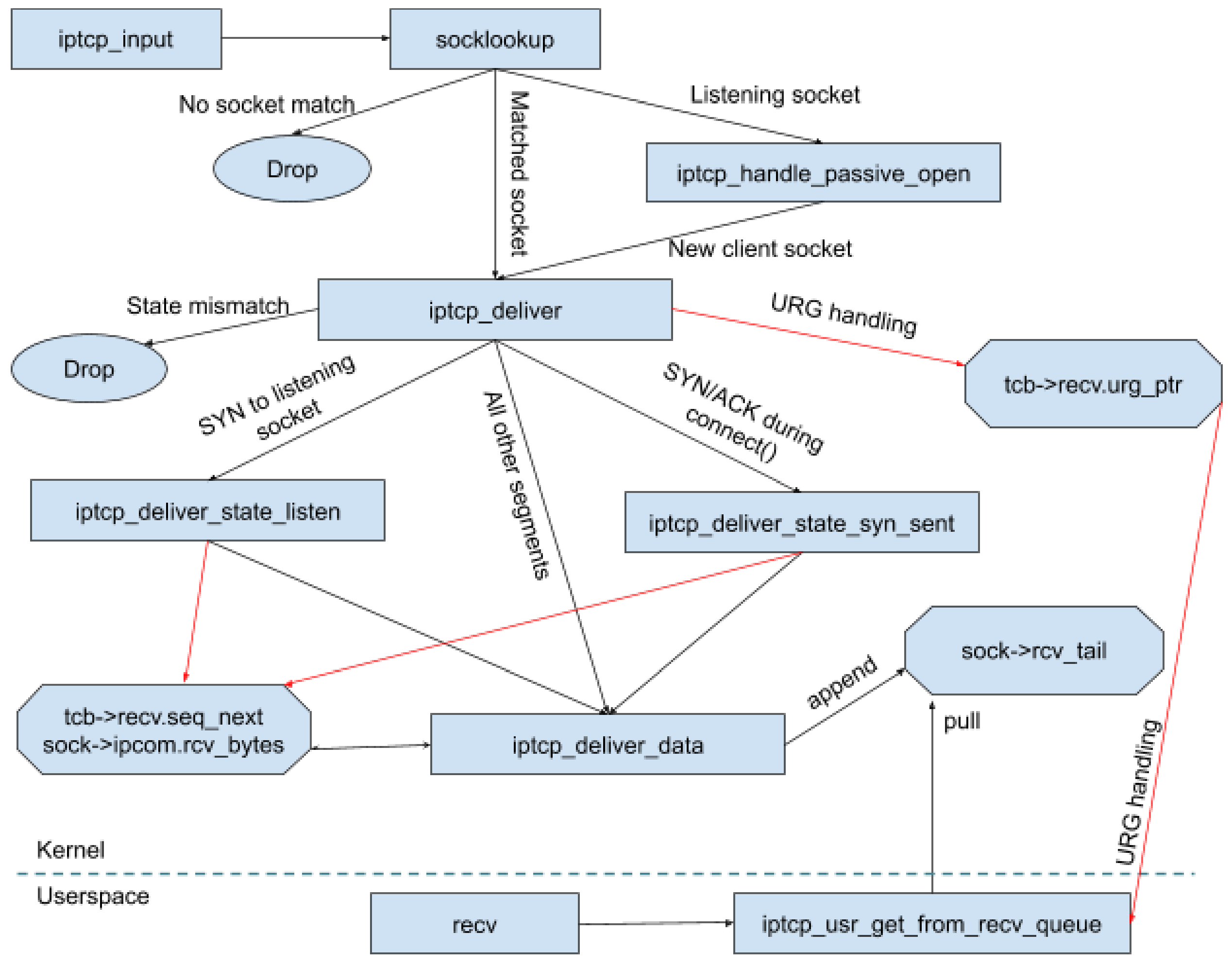
The function *iptcp_input* tries to match every incoming segment to an existing TCP socket, using the *socklookup* callback. A segment may also match a **listening** socket, if the 4-tuple **doesn't** match an existing socket, but the destination IP and port do match those of a bound listening socket. In this case, an additional call to *iptcp_handle_passive_open* performs extra logic, ending up with the creation of a **new** client socket for this connection, that does match the 4-tuple. This socket then becomes the new matched socket from that point on. All validations of the first SYN packet are done by *iptcp_handle_passive_open,* for example - drop any SYN packet that is also a FIN.

After some additional checks, the segment is passed on to *iptcp_deliver*, which handles a segment in the context of a TCP socket (*tcb*):

```
int __fastcall iptcp_deliver(_DWORD *a1)
{
  if ( tcb->state == 2 )  {       // LISTEN_STATE
    result = iptcp_deliver_state_listen(...);
  } else if ( tcb->state == 3 ) { // SYN_SENT_STATE
    result = iptcp_deliver_state_syn_sent(...);
  } else {
    result = iptcp_deliver_state_syn_rcvd_or_higher(...);
  }
  ...
  if ( pkt->flags & 0x2000 ) { // Check if URG flag in TCP pkt
    tcb->urg_ptr = p->seg.seq_start + htons(pkt->urg_ptr);
    tcb->flags |= 0x80000; // TCB_STATE_URG_RECEIVED
    ...
  }
  if ( pkt->flags & 0x100 ) { // Check if FIN flag in TCP pkt
    ...
    // Handle the FIN bit, depending on the current TCB state
    switch ( tcb->state ) {
      ...
      default:
        return -22;
    }
  }
  ....
  // Process the segment data
  result = iptcp_deliver_data(v2);
  if ( tcb->state != newState ) {
     // If the state has changed, apply it to the tcb
     ...
     result = iptcp_change_state(tcb->sock, newState);
  }
  ...
}
```

Simplified decompilation of *iptcp_deliver*

First, this function checks the state of the socket. If it was just created, the first segment will be handled by *iptcp_deliver_state_listen* or *iptcp_deliver_state_syn_sent* in a special way. The *Initial Sequence Number* from the peer will initialize *tcb->recv.seq_next,* which is the *next in-order sequence number* for this socket. After additional handling, such as handling the URG and FIN flags, the segment may be passed into *iptcp_deliver_data*. This is where the *TCP window* is managed. Once in-order data becomes available, it will eventually be appended to the *sock->rcv_tail* list, which is the list *recv()* calls pull data from. An overview of the flow, starting at the *iptcp_input* function, and ending in a user application calling *recv()*, looks like this:



Call flow from TCP segments to recv calls

From the userland code, a call to *recv()* will use *iptcp_usr_get_from_recv_queue* to pull data from the *sock->rcv_tail* queue. Prior to that, however, this function also handles any *Urgent Data* that has been received as well, accessing the state variable *tcb->recv.urg_ptr*. As seen in the decompilation snippet above, this state variable is updated in *iptcp_deliver* each time a segment with an URG flag is received.

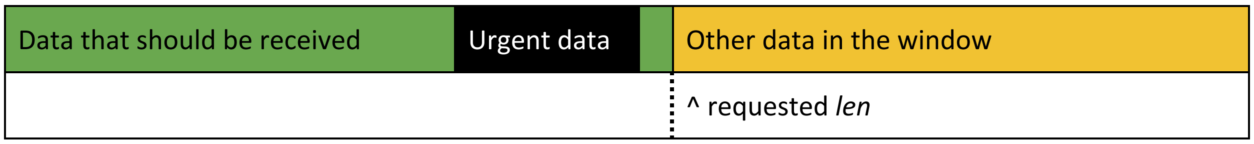
TCP Urgent data issues in the IPnet stack

When *iptcp_deliver* receives a segment with the URG flag set, the *FLAG_RECEIVER_URG* flag in *tcb->flags* is set and the *tcb->recv.urg_ptr* value is calculated as the offset in the TCP window where the urgent data begins. Later, this value is used by *iptcp_usr_get_from_recv_queue* when called from a *recv()* call. This value is important for all *recv* calls and not just for *MSB_OOB* calls, as any *Urgent Data* present in the segment **must not** be returned from a regular *recv* call. Therefore, this value is used to know which bytes to **drop** from the returned buffer. Below is the code responsible for this flow:

```
// Urgent Data is present, but not requested with the MSG_OOB flag
if ((int32)(tcb->recv.urg_ptr - len +
            tcb->recv.seq_next - sock->ipcom.rcv_bytes) <= 0)
{
    // Calculate the urgent data offset inside the window, in order to
    // copy data up to, but not including the urgent data
    len = tcb->recv.urg_ptr - 1 - tcb->recv.seq_next + sock->ipcom.rcv_bytes;
}
```

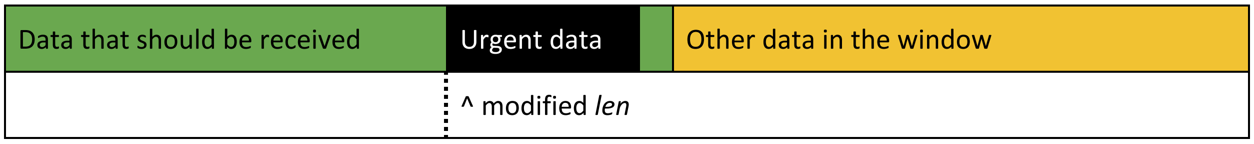
Simplified decompilation, from *iptcp_usr_get_from_recv_queue*
when handling a regular *recv()* with URG data present

This code changes the *len* variable in case *Urgent Data* was inserted into the *TCP window* so that the current *recv()* call is affected. As illustrated:



The *recv()* call requested a read of size ***len*,** but this would include the Urgent Data

The purpose of the code above is to **shorten** the *len* in this case, so that the user receives less data than requested, without any OOB data that was not requested:



We'll henceforth refer to the above *len* shortening calculation as the *urgent data offset calculation*.

As mentioned earlier, due to the various intricacies of the Urgent Pointer mechanism, VxWorks includes a *RFC-1122 compatible* mode, and a non-compatible mode - where the Urgent Data points to ±1 of the calculated urgent pointer. However, by default, VxWorks **doesn't** support RFC-1122 compatible mode, thus the above *urgent data offset calculation* subtracts 1 from the *tcb->recv.urg_ptr* to calculate where the Urgent Data starts.

All of the 3 state variables referenced in the code above are unsigned 32 bit integers, as is the *len* variable to which the above calculation is written to. We have discovered 4 different vulnerabilities, in various code flows, all eventually cause the calculation of the *urgent data offset* to underflow causing the *len* variable to become a huge unsigned integer.

The first of these vulnerabilities (Urgent Pointer = 0) results in an off-by-one calculation, and the other three variants are caused by various state confusion states that lead the *tcb->recv.urg_ptr* and *tcb->recv.seq_next* variables to be in an inconsistent state with one another. The *urgent data offset calculation* works under the assumption that the Urgent Pointer (*tcb->recv.urg_ptr*) is always in front of the sequence number that the TCP window begins with, calculated by *tcb->recv.seq_next -*

sock->ipcom.rcv_bytes. Once this assumption breaks, the calculation can result in an integer underflow, and *len* will become a huge unsigned integer.

Since *len* is the limitation passed by the user to a *recv()* call, this effectively allows an attacker to *disable* the length check altogether. This renders any userspace code using *recv* on a TCP socket vulnerable to an overflow of the local receive buffer. For example:

```
char buf = 0;
...
recv(socket_fd, &buf, 1);
```

This usually safe code that receives only 1 byte into a stack variable now becomes a stack overflow. The overflow length is controlled by the attacker, that chooses how much data to put into the *TCP window*. Despite the *len* becoming a huge number, the amount of data actually written will still be limited by the amount of data available in the window. So, as long as the buffer passed to *recv()* is **smaller** than the *TCP window* of the established connection, an overflow will occur. The buffer passed to *recv()* may be allocated either in the stack, heap or the application's global data section which means an overflow of this buffer will result in varying effects once triggered. The exploitation process of such an overflow will have to vary accordingly.

Each of the discovered Urgent Pointer vulnerabilities affects a different set of VxWorks' versions, but combined they span from version 6.5 and above. The following sections will detail the four different code flows in which the above overflow can be triggered.

TCP Urgent pointer = 0 integer underflow (CVE-2019-12255)

As shown previously, a TCP connection's Urgent Pointer is set in the variable *tcb->recv.urg_ptr* in a code flow inside *iptcp_deliver*. In VxWorks' versions 6.9.3 and below this code flow looks like this:

```
if ( pkt->flags & 0x2000 ) { // Check if URG flag in TCP pkt
  tcb->urg_ptr = p->seg.seq_start + htons(pkt->urg_ptr);
  tcb->flags |= 0x80000; // TCB_STATE_URG_RECEIVED
  ...
}
```

If the *urgent_pointer* field in the received TCP segment header is set to 0, *tcb->recv.urg_ptr* will be equal to *p->seg.seq_start,* which is the *Sequence Number* of the received segment. Then, when the user of the socket will perform a *recv()* operation on the socket, the code presented in the section above (inside *iptcp_usr_get_from_recv_queue*) will be triggered.

The condition of the **if** in the function *iptcp_usr_get_from_recv_queue* shown in the section above can thus be re-written by substituting for equivalent values:

```
(int32)(tcb->recv.urg_ptr - len + tcb->recv.seq_next - sock->ipcom.rcv_bytes) <= 0
⇔ (int32)(p->seg.seq_start + 0 - len - p->seg.seq_start) <= 0
⇔ (int32)(0 - len) <= 0
```

Note that *p->seg.seq_start* equals *tcb->recv.seq_next - sock->ipcom.rcv_bytes* for the last received segment. Therefore this condition is always true when the *urgent_pointer* was set to 0 in the last received segment. Then, *len* is supposed to be shortened by the *urgent data offset calculation* (as mentioned in the above section), however:

```
  tcb->recv.urg_ptr - 1 - tcb->recv.seq_next + sock->ipcom.rcv_bytes
⇔ p->seg.seq_start - 1 - (p->seg.seq_start)
⇔ -1
```

The function tries to evaluate the offset to an Urgent Data that is out of bounds (0), which results in -1. Since *len* is an unsigned integer, this means it will now equal 0xffffffff. As mentioned above, this causes the length constraint **set by the user** in the *recv()* call to be ignored, resulting in turn in a copy of **all the available** data from the TCP window to the user supplied buffer.

In VxWorks' version 6.9.4 an additional validation on the Urgent Pointer was added in *iptcp_deliver*:

```
    if ( flags & 0x2000) { // Check if URG flag in packet
      tcb_flags = tcb->flags;
      if ( tcb_flags & 0x80000) { // Check if URG was already received
         ...
      } else {
        urgent_pointer = input_tcp_header->urgent_pointer;
        error_retval = 1;
        if ( !urgent_pointer )
          return error_retval;
```

Decompilation snippet from *iptcp_deliver*

This fixes exactly the issue detailed above, preventing *urgent_pointer* from being set to 0. However, unfortunately, this change was not considered as a security fix, and wasn't backported to prior versions of VxWorks. All real world products we've examined, that were running versions of VxWorks 6.9.3 or prior, were vulnerable to this vulnerability.

TCP Urgent Pointer state confusion caused by malformed TCP AO option (CVE-2019-12260)

While the *Urgent Pointer = 0* bug from above was fixed since VxWorks version 6.9.3, the *iptcp.c* module had undergone other refactoring in the meantime. Some of the new features that were added, like the handling of a new AO TCP option, introduced even deeper bugs to the code.

In VxWorks versions **above** 6.9.3, support for the TCP AO option (Authentication Option, RFC-5925) was added. This appears to be included by default, and is indeed supported by the VxWorks images of products we've examined. The vulnerability presented below does not depend on the TCP AO to be actually enabled or used, since it is always parsed by the TCP module.

The following flow will occur inside *iptcp_input* for an incoming **SYN** packet from a client that is establishing a new connection:

```
  ...
  new_client_sock = iptcp_handle_passive_open(src_ip, dst_ip, &params);
  ...
  tcb = new_client_sock->tcb;
  ...
  tcp_options_size = ((tcp_segment_offset_flags & 0xF0) >> 2) - 20;
  tcp_options_ptr = &tcp_header_ptr[1];
  if ( tcp_options_size > 0 )
  {
    index = 0;
    while ( 1 )
    {
      opt_ptr = &tcp_options_ptr[index];
      opt_kind = *opt_ptr;
      ...
    }
    if ( opt_kind == TCP_OPTION_AO )
    {
      opt_len = opt_ptr[1];
      if ( opt_len <= 3u )
      {
        iptcp_ao_log(6, "discard the segment since TCP-AO...", ...);
        goto FreeAndExit;
...
FreeAndExit:
...
    ipcom_pkt_free(pkt);
    return 0;
```

Snippet from a decompilation output from a binary image, inside *iptcp_input*

The support for the TCP AO option was added to the familiar *iptcp_input* resulting in some changes to the function. The function *iptcp_handle_passive_open* will be called when a **SYN** arrives on a listening socket, and a new socket object will be created for the incoming connection. This socket will now be added to the destination cache, so that it will match any new TCP segment arriving from the same TCP connection 4-tuple (IP src/dst and Port src/dst). Immediately thereafter the code will look for an AO Option (TCP_OPTION_AO) in the TCP option header and attempt to parse it if found.

Now, assume an attacker added a **malformed** TCP_OPTION_AO option to that very first SYN packet, like some 1 byte value for the option - which is too short, per the condition *opt_len <= 3*. The check will fail, and *iptcp_input* will simply drop the packet and return. However, the new socket that was created by *iptcp_handle_passive_open* **will not be destroyed**.

At this point, a socket object that **will match** any incoming TCP segment on the connection's 4-tuple **still exists**, however, since the **SYN** packet was not fully processed the new socket will remain in the default *LISTEN_STATE* state. Now an attacker can send a **new** (#2) SYN packet, which will **not** be processed by *iptcp_handle_passive_open*, since it arrived to an already existing client socket (and **not** to the listening socket). Since *iptcp_handle_passive_open* doesn't process this new SYN packet, the following validations are **not** performed by it:

```
if ((p->seg.flags_n & TCP_SYN_FLAG) == 0) ||
    (p->seg.flags_n & (TCP_FIN_FLAG | TCP_URG_FLAG | TCP_PSH_FLAG)) != 0) {
    // Validate that SYN packet doesn't have FIN, URG, or PSH flags turned on.
    ...
    return 0;
```

Validations of initial SYN packet, done in *iptcp_handle_passive_open*

Now assume that the attacker has set the flags **URG** and **FIN** together with **SYN** in this new (#2) SYN packet. This new packet will bypass the checks above as an otherwise valid packet, continue the flow and arrive to *iptcp_deliver*. In there it will be handled according to the *tcb* state:

```
  if ( tcb->state == 2 )  {       // LISTEN_STATE
    result = iptcp_deliver_state_listen(...);
  } else if ...
```

Following this, the function *iptcp_process_syn* will be called by *iptcp_deliver_state_listen*, performing the following:

```
   tcb->recv.seq_next = p->seg.seq_start;
```

The recv.seq_next of the socket will now be set to the *Initial Sequence Number* chosen by the attacker in this **SYN/URG/FIN** (#2) packet. Let that number be denoted as *sequence_a*.

The flow continues in *iptcp_deliver*, and immediately enters this (familiar) flow:

```
  if ( pkt->flags & 0x2000 ) { // Check if URG flag in TCP pkt
    tcb->urg_ptr = p->seg.seq_start + htons(pkt->urg_ptr);
    tcb->flags |= 0x80000; // TCB_STATE_URG_RECEIVED
    ...
  }
```

Therefore, assuming *tcp_hdr->urgent_pointer = 1* (a valid non-zero value), the variable *tcb->recv.urg_ptr* is assigned the value *p->seg.seq_start + 1 = sequence_a + 1*.

The flow continues in *iptcp_deliver*, but immediately encounters the following check:

```
if ( pkt->flags & 0x100 ) { // Check if FIN flag in TCP pkt
  ...
  // Handle the FIN bit, depending on the current TCB state
  switch ( tcb->state ) {
    ...
    default:
      return -22;
  }
}
```

Since the state of the socket is still not valid for a **FIN** to be received, this check fails and an error is returned from *itpcp_deliver* back to *iptcp_input*. The state of the socket (*tcb->state*) is thus **not changed** yet again, and remains *LISTEN_STATE*! As for *iptcp_input*, it now simply drops this packet due to the error, but the socket itself **still remains alive**. Notably, the *TCB_STATE_URG_RECEIVED* is left **set** on the socket, and the *recv.urg_ptr* retains the value *sequence_a + 1*.

Finally, at this point a **valid** (#3) **SYN** packet is sent to this socket on the 4-tuple, with an *Initial Sequence Number* value *sequence_b*. Assume that *sequence_b = sequence_a + 1000000*. This is a fully valid SYN packet, and it will be handled by the code as intended. A SYN/ACK will be sent back to the attacker, and the attacker will respond with an ACK to finalize the handshake. Additionally, the attacker may include up to 64k bytes of data with this ACK segment, which will be added to the TCP receive window of the socket.

Only now will the state of the socket become *ESTABLISHED_STATE*, and a **user** waiting on an *accept()* call will be handed the client socket. Then the user will likely call *recv()* on the socket. At this point, the familiar code in *iptcp_usr_get_from_recv_queue* will be executed:

```
...
if (tcb->flags & 0x80000))) { // TCB_STATE_URG_RECEIVED
  ...
  if (iptcp_at_mark(sock)) {
    ...
  } else if ((int32)(tcb->recv.urg_ptr - len +
                tcb->recv.seq_next - sock->ipcom.rcv_bytes) <= 0) {
    // Calculate the urgent data offset inside the window, in order to
    // copy data up to, but not including the urgent data
    len = tcb->recv.urg_ptr - 1 - tcb->recv.seq_next + sock->ipcom.rcv_bytes;
  }
```

The *TCB_STATE_URG_RECEIVED* is **still set** at this point, and the *recv.urg_ptr* retains its value too. Therefore, the **else-if** condition is checked, and can now be substituted with its matching values:

```
(int32)(tcb->recv.urg_ptr - len + tcb->recv.seq_next - sock->ipcom.rcv_bytes) <= 0
⇔ (int32)(sequence_a + 1 - len - sequence_b) <= 0
⇔ (int32)(sequence_a + 1 - len - sequence_a + 1000000) <= 0
⇔ (int32)(-len - 999999) <= 0
```

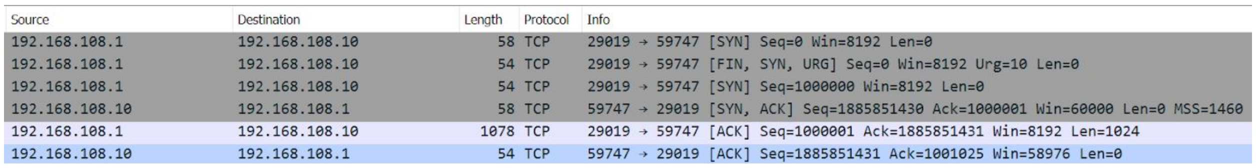
Therefore the check is passed, and *len* will be set by the *urgent data offset calculation* as such:

```
  tcb->recv.urg_ptr - 1 - tcb->recv.seq_next + sock->ipcom.rcv_bytes
⇔ sequence_a + 1 - 1 - (sequence_a + 1000000)
⇔ -1000000
```

Therefore, *len* will now equal -1000000, and since *len* is an unsigned 32-bit integer, it will now equal a very large number, voiding any user defined restrictions. Similar to the previous Urgent Pointer vulnerability described above, this will result in trivial overflows in any code that performs *recv()* on this TCP socket.

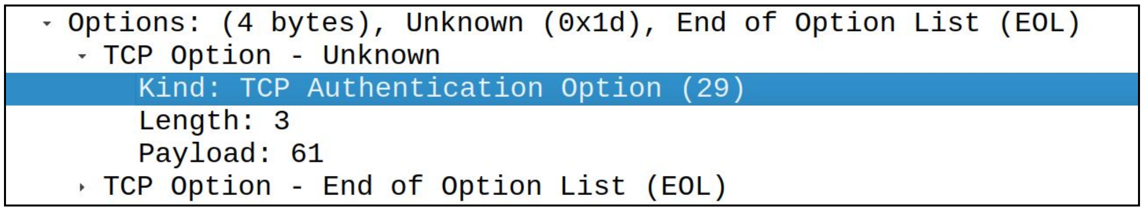
A 5-way-handshake

The following wireshark capture shows the TCP-AO state confusion attack described above. The target device (192.168.108.10) has a server listening on TCP port 59747:



Source	Destination	Length	Protocol	Info
192.168.108.1	192.168.108.10	58	TCP	29019 → 59747 [SYN] Seq=0 Win=8192 Len=0
192.168.108.1	192.168.108.10	54	TCP	29019 → 59747 [FIN, SYN, URG] Seq=0 Win=8192 Urg=10 Len=0
192.168.108.1	192.168.108.10	54	TCP	29019 → 59747 [SYN] Seq=1000000 Win=8192 Len=0
192.168.108.10	192.168.108.1	58	TCP	59747 → 29019 [SYN, ACK] Seq=1885851430 Ack=1000001 Win=60000 Len=0 MSS=1460
192.168.108.1	192.168.108.10	1078	TCP	29019 → 59747 [ACK] Seq=1000001 Ack=1885851431 Win=8192 Len=1024
192.168.108.10	192.168.108.1	54	TCP	59747 → 29019 [ACK] Seq=1885851431 Ack=1001025 Win=58976 Len=0

Wireshark capture of the attack packets



```
▾ Options: (4 bytes), Unknown (0x1d), End of Option List (EOL)
  ▾ TCP Option - Unknown
      Kind: TCP Authentication Option (29)
      Length: 3
      Payload: 61
  ▸ TCP Option - End of Option List (EOL)
```

TCP options field of the first packet from the capture above

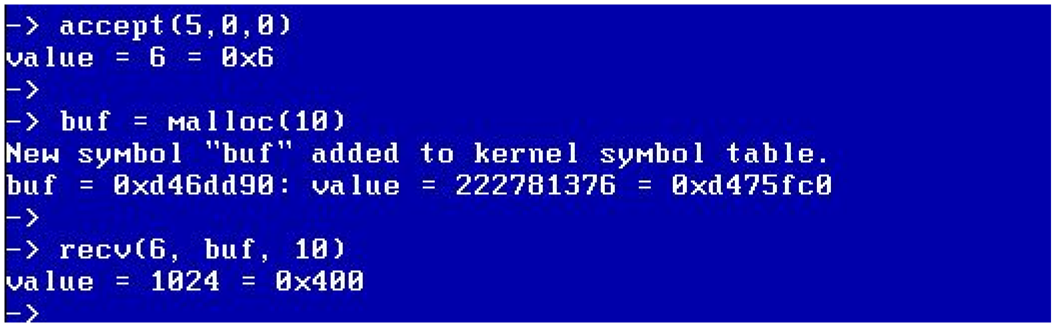
As you can see, the first 3 packets from the attacker to the target have the **SYN** flag set:

1) The first packet has a malformed TCP-AO option (shown in the second capture)
2) The second packet has *Seq = 0* and the URG, FIN flags are set. The *Urgent Pointer* is also set to 10 (an arbitrary non-zero value)
3) The third packet has *Seq = 1000000* and is otherwise valid

A non-vulnerable TCP stack should have returned an RST packet for each of the first 2 packets. However, in the case of a vulnerable IPnet stack, these packets are handled in an incorrect fashion, resulting in a state confusion issue. The end result is an open (connected) socket, which has *tcb->recv.urg_ptr* set to 10 and *tcb->recv.seq_next* set to 1000000. This will cause the *urgent data offset calculation* to underflow during *recv()* calls, as explained in the sections above.

Following the first 3 packets, the target responds with a SYN/ACK, indicating the reception of the 3rd SYN packet. The 5th packet is an ACK sent by the attacker to finish the handshake. This ACK packet has a payload attached, with a length of 1024. At this point any subsequent *recv()* call on the socket will result in 1024 bytes written into the user buffer, regardless of the *len* passed by the user.

If the TCP server on port 59747 performs a *recv()* of a shorter length, a memory corruption will occur. Moreover, the attacker controls the overflow data, as it's the data that was sent in the ACK packet.



```
-> accept(5,0,0)
value = 6 = 0x6
->
-> buf = malloc(10)
New symbol "buf" added to kernel symbol table.
buf = 0xd46dd90: value = 222781376 = 0xd475fc0
->
-> recv(6, buf, 10)
value = 1024 = 0x400
->
```

Series of C-interp calls simulating a listening server on a VxWorks BSP VM

The screen capture above is from a VxWorks BSP VM running v6.9.3. Here, *fd = 5* is the listening socket of the server which was opened prior. After the attack is performed, the server *accepts()* the attacker connection, resulting in an open client socket (*fd = 6*). Then a *recv()* of length **10** is performed. However, 1024 bytes are written (!) as indicated by the return value of *recv* which results (in this case) in a heap overflow.

TCP Urgent Pointer state confusion during connect() to a remote host (CVE-2019-12261)

The previous code flows where achievable when a target device was acting as a TCP server, and an attacker initiated a connection with that server. An additional Urgent Pointer vulnerability variant exists when a VxWorks device running version 6.7 or above creates an outbound TCP connection. In this state, an exploitable state confusion is possible during a *connect()* call on a TCP socket, after the target device sends a **SYN** packet to a remote TCP port. If an attacker responds to this SYN with a specially crafted SYN/ACK, a similar issue with urgent pointer handling arises as in the previous section of this document.

To exploit the issue, an attacker needs to be in either of the following positions:

1) Coerce the target device to create a TCP connection to a malicious host.
2) Be in a Man-in-the-middle (MiTM) position between the target device and a legitimate host that it connects to, which can be easily achieved for an attacker on the LAN.

After a *connect()* call is made by the user application, a SYN packet is sent to the peer, and the socket enters the *SYN_SENT_STATE* state. At this point, a SYN/ACK packet is expected to arrive to *iptcp_deliver*. When a packet arrives in this state, the *iptcp_deliver_state_syn_sent* will be called:

```
if (p->seg.flags_n & 0x1000) { // TCP_ACK_FLAG
    ...
    *newState = ESTABLISHED_STATE;
}
...
if (p->seg.flags_n & 0x200) { // TCP_SYN_FLAG
    ...
    iptcp_process_syn(p);
    return 2;
}
```

If the received packet indeed had the SYN and ACK flags set, *iptcp_process_syn* is called, and *tcb->recv.seq_next* is assigned the value *p->seg.seq_start* which is the Initial Sequence Number inside the SYN/ACK packet. This value is attacker controlled, and shall be denoted as *sequence_a*. Note that **new_state* does not actually set the *tcb* state yet. This is supposed to happen later, inside iptcp_deliver.

Now, assume that the attacker has added the **FIN** and **URG** flags to the SYN/ACK packet, creating a **SYN/ACK/FIN/URG** (#1) response instead of a regular SYN/ACK. The code flow shown above happens exactly the same way as with a regular SYN/ACK. There are no checks against the existence of the FIN and URG flags in *iptcp_deliver_state_syn_sent* (or before).

The rest of the flow inside *iptcp_deliver* will set the *tcb->recv.urg_ptr* (which is set to *sequence_a + 1*) and then abort further processing due to the unexpected FIN. Therefore, the state of the *tcb* is again **not** updated, and thus remains *SYN_SENT_STATE*.

An attacker can now send **another** SYN/ACK (#2) packet. This time it will be a valid SYN/ACK, but with an Initial Sequence Number denoted as *sequence_b*. Additionally, the attacker may include up to 64k bytes of data with this SYN/ACK segment, which will be added to the TCP receive window of the socket. Only now the state of the socket will become *ESTABLISHED_STATE*, and the *connect()* call will return.

However, at this point, there is again a disparity between the value of *tcb->recv.urg_ptr* (derived from *sequence_a*) and *tcb->recv.seq_next* (derived from *sequence_b*). To reiterate, this will cause any *recv()* call on the socket to corrupt memory in an attacker controlled fashion, resulting in RCE.

TCP Urgent Pointer state confusion due to race condition (CVE-2019-12263)

The last Urgent Pointer vulnerability variant we discovered is a race condition affecting all VxWorks' devices that use the IPnet stack (v6.5 and above) and can result in a state confusion of the urgent pointer. Similar to the previous examples which demonstrated the result of this type of state confusion, this race condition could result in a memory corruption of a user task buffer that can lead to remote code execution triggerable on a target device acting either as a TCP server, or a TCP client.

The race condition results from the fact that various variables represent the state of the urgent pointer as a whole but might be individually changed by the kernel task (*tNet0*) while a user task is accessing them (in *iptcp_usr_get_from_recv_queue*). This may occur if the user task is running in a different priority than the kernel task, or if SMP (Symmetric Multi-Processing) is in use in a multi-core environment. There is no lock or mutex to prevent a race condition of this nature from occurring.

The various variables that together represent the state of the urgent pointer can be seen in this familiar code in *iptcp_usr_get_from_recv_queue*:

```
...
if (tcb->flags & 0x80000))) { // TCB_STATE_URG_RECEIVED
  ...
  if (iptcp_at_mark(sock)) {
    ...
  } else if ((int32)(tcb->recv.urg_ptr - len +
              tcb->recv.seq_next - sock->ipcom.rcv_bytes) <= 0) {
    // Calculate the urgent data offset inside the window, in order to
    // copy data up to, but not including the urgent data
    len = tcb->recv.urg_ptr - 1 - tcb->recv.seq_next - sock->ipcom.rcv_bytes;
  }
```

This code assumes that the *TCB_STATE_URG_RECEIVED* is set in conjunction to setting the *tcb->recv.urg_ptr* variable. It also depends on *tcb->recv.seq_next* and *sock->ipcom.rcv_bytes* being set at the same time, and that *tcb->recv.urg_ptr* is set in conjunction with these variables as well.

For example, if the *len* calculation inside is **preempted** between fetching the *sock->ipcom.rcv_bytes* value and the *tcb->recv.seq_next* value, any new segment received in the meantime will increase the difference between the two fetched values in this function arbitrarily. This will enable an attacker to underflow *len*, causing the vulnerable condition described prior. Other examples of this race condition that will result in this underflow are also possible.

Such a race condition is possible if the priority of the task that performs the *recv()* call is lower than the IPnet task's priority, so that it will be preempted immediately once a new packet is received from the network. In any case, it is also possible on multi-core SMP systems regardless of priorities.

Heap overflow in DHCP Offer/ACK parsing in VxWorks' DHCP client (ipdhcpc)

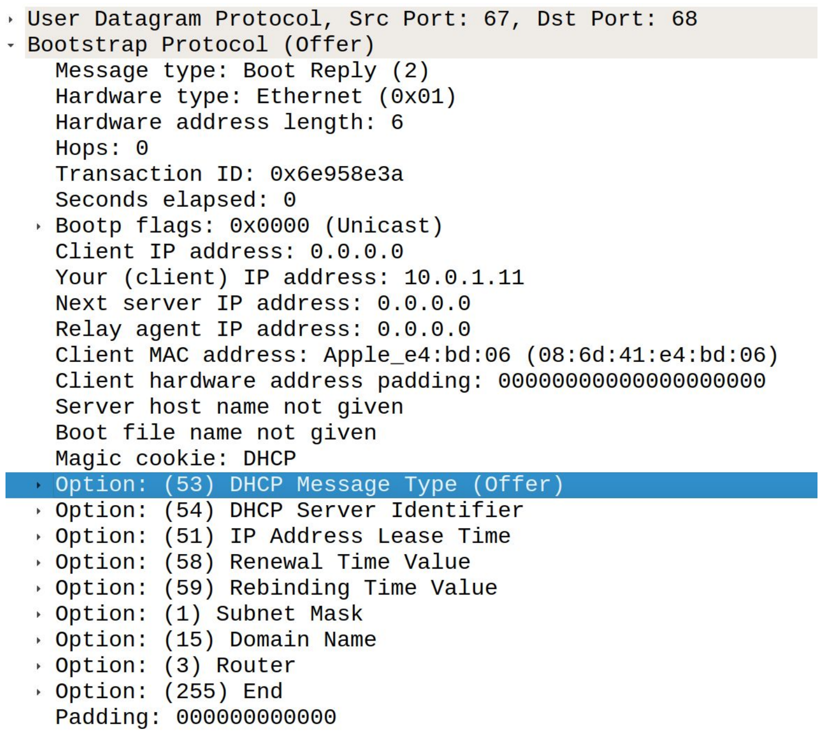
TL;DR

A heap overflow vulnerability exists in VxWorks' DHCP client when parsing incoming DHCP Offer and ACK packets. An attacker on the LAN can trigger this vulnerability by sending a specially crafted DHCP Offer packet to a target device that has sent a DHCP Discover packet or by sending a DHCP ACK packet in response to a target device that has sent a DHCP Request packet. This overflow contains attacker controlled bytes, and can lead to remote code execution.

Background - DHCP options

The DHCP protocol allows automatic network configurations for all devices on a LAN. Prior to DHCP, each machine had to be configured manually with an IP address, subnet mask and default gateway IP. DHCP utilizes a central server which answers to *"DHCP request"* broadcasts with *Offers* that contain network configurations.

Every device on the network that desires to be configured by the central DHCP server, runs a local DHCP client daemon. This daemon sends a broadcast and waits for the configuration to arrive from a server. This communication takes place over UDP ports 67 and 68. The structure of an *Offer* packet that arrives as a response to a client is illustrated below:



```
▸ User Datagram Protocol, Src Port: 67, Dst Port: 68
▾ Bootstrap Protocol (Offer)
    Message type: Boot Reply (2)
    Hardware type: Ethernet (0x01)
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0x6e958e3a
    Seconds elapsed: 0
  ▸ Bootp flags: 0x0000 (Unicast)
    Client IP address: 0.0.0.0
    Your (client) IP address: 10.0.1.11
    Next server IP address: 0.0.0.0
    Relay agent IP address: 0.0.0.0
    Client MAC address: Apple_e4:bd:06 (08:6d:41:e4:bd:06)
    Client hardware address padding: 00000000000000000000
    Server host name not given
    Boot file name not given
    Magic cookie: DHCP
  ▸ Option: (53) DHCP Message Type (Offer)
  ▸ Option: (54) DHCP Server Identifier
  ▸ Option: (51) IP Address Lease Time
  ▸ Option: (58) Renewal Time Value
  ▸ Option: (59) Rebinding Time Value
  ▸ Option: (1) Subnet Mask
  ▸ Option: (15) Domain Name
  ▸ Option: (3) Router
  ▸ Option: (255) End
    Padding: 000000000000
```

Note the variable length options list at the end of the packet

An attacker on the LAN will also receive all *DHCP discover* broadcasts sent on the network, and will be able to respond to them, as the protocol includes no authentication.

The VxWorks ipdhcpc DHCP client daemon has a heap overflow vulnerability since version 6.7 and above in the parsing of those *DHCP Offer* packets. The vulnerability can be triggered by an attacker on the LAN who responds with a specially crafted DHCP Offer packet to a target device that has sent a DHCP Discover packet. The same also applies for the DHCP ACK packet, in response to a DHCP Request packet.

The vulnerability (CVE-2019-12257)

The DHCP Offer (and ACK) packet ends with a variable length options array. The ipdhcpc daemon allocates space on the heap for the incoming options inside *ipdhcpc_handle_malloc*:

```
...
handle->info.options = ipcom_malloc(ipdhcpc.max_message_size - 264);
```

The buffer is thus allocated to the size *ipdhcpc.max_message_size - 264*. Later, when the daemon is waiting to receive a response packet, a *recvfrom()* call is performed inside the main loop of *ipdhcpc_daemon*:

```
ipdhcpc.in_pkt_len = ipcom_recvfrom(ipdhcpc.fd,
                                    ipdhcpc.in_pkt,
                                    ipdhcpc.max_message_size,
                                    ...);
```

A packet of at most *ipdhcpc.max_message_size* bytes is received here, and its received length is set in *ipdhcpc.in_pkt_len*. The flow continues to parse the packet, arriving in *ipdhcpc_reply_input*, where the length of the incoming options is calculated based on *ipdhcpc.in_pkt_len*:

```
  handle->priv->in_optlen = ipdhcpc.in_pkt_len - 240;
```

Later, the flow continues to *ipdhcpc_offer_input* (or *ipdhcpc_ack_input*), where the following *memcpy()* will occur:

```
handle->info.optlen = handle->priv->in_optlen;
ipcom_memcpy(handle->info.options,
             &ipdhcpc.in_pkt->options[4],
             handle->info.optlen);
```

Note that the maximum value for *ipdhcpc.in_pkt_len* is *ipdhcpc.max_message_size*. Therefore the maximum value for *handle->priv->in_optlen* can be *ipdhcpc.max_message_size* - ***240***, which is **greater** than the space allocated for *handle->info.options*, chosen to be *ipdhcpc.max_message_size* - ***264***. This results in a 24 byte heap overflow with attacker controlled data.

The structure of *ipcom_malloc* heap blocks consists of 16 bytes of metadata, and an 8 byte alignment trailer. Therefore, 24 bytes are exactly enough to corrupt the metadata of the next block, making RCE achievable.

Interestingly, a similar vulnerability in the parsing of DHCP options was found in the Windows 10 DHCP client recently, known as CVE-2019-0547. A good writeup is available here.

Five Logical errors, DoS and Information Leak vulnerabilities

In addition to the six RCE vulnerabilities described in previous sections, we have also discovered five additional vulnerabilities that result in either logical errors, denial of service, or information leaks.

These vulnerabilities where found in various subsystems of the TCP/IP stack - TCP, IGMP, DHCP, and even in the very ancient Reverse ARP (RARP) protocol. The following sections will describe these vulnerabilities.

TCP connection DoS via malformed TCP options (CVE-2019-12258)

This issue affects all VxWorks' versions that use IPnet (version 6.5 and up). An attacker can send a specially crafted TCP packet with the 4-tuple of an existing connection, but without knowing the sequence numbers of that connection, which will cause the connection to drop. This ability can allow an attacker a denial-of-service for any TCP connection to or from an affected VxWorks device.

The specially crafted TCP packet contains illegal IP options, causing the function *iptcp_process_options* to fail:

```
int __fastcall iptcp_process_options(_DWORD *a1, tcp_hdr_t *tcp_hdr) {
    ...
    v10 = 0;
    while (v10 < tcp_options_length) {
        opt = &tcp_hdr->options[v10];
        ...
        switch (opt->type)
        {
        case 2: // TCP_MSS_OPTION_TYPE
            if (opt->length != 4)
                // If MSS option isn't 4 bytes length - break the flow.
                return -22;
```

Simplified decompilation snippet from *iptcp_process_options*

The above function may fail and return a negative value error code in various code-flows. For example, if a TCP packet with an *TCP_MSS_OPTION_TYPE (2)* option is received with an option length that is not 4 bytes the function will fail with *-22*.

This function is called by *iptcp_input* (described in depth in a previous section) and when it fails the TCP connection on which it was received is dropped by *iptcp_send_reset*:

```
    v32 = iptcp_process_options(&v68, tcp_hdr);
    if (v32 < 0)
    {
      iptcp_send_reset(...);
      return 0;
    }
```

Inside iptcp_input

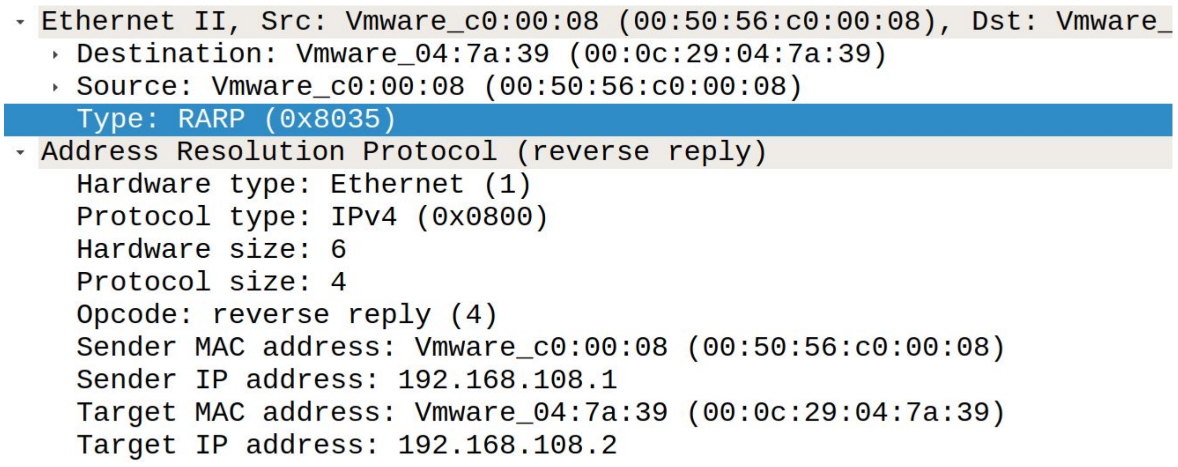
So, by sending a TCP packet with an invalid TCP options header, a remote attacker can cause the TCP connection to disconnect, based solely on the 4-tuple of that connection.

Handling of unsolicited Reverse ARP replies (CVE-2019-12262)

One of the most bizarre vulnerabilities we found is a logical error vulnerability in the Reverse ARP (RARP) protocol. An attacker on the local LAN can send unsolicited RARP reply packets to a target device (by its MAC). These in turn will allow him to add IPv4 addresses to the interface that receives them on the target device, which can lead to various denial-of-service attacks.

RARP is an old protocol (from 1984) that preceded DHCP for the purposes of automatic network address configuration. Similarly to DHCP, a client broadcasts a request for an address to be assigned, and listens for a response that contains its assigned address.

In reality, no modern network or device should support RARP since it's incredibly obsolete and practically unused. While the IPnet stack does support RARP, it doesn't attempt to send a RARP request by default. However, responses are still handled.



```
▾ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_
  ▸ Destination: Vmware_04:7a:39 (00:0c:29:04:7a:39)
  ▸ Source: Vmware_c0:00:08 (00:50:56:c0:00:08)
    Type: RARP (0x8035)
▾ Address Resolution Protocol (reverse reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reverse reply (4)
    Sender MAC address: Vmware_c0:00:08 (00:50:56:c0:00:08)
    Sender IP address: 192.168.108.1
    Target MAC address: Vmware_04:7a:39 (00:0c:29:04:7a:39)
    Target IP address: 192.168.108.2
```

RARP reply packet that assigns the address 192.168.108.2

The RARP protocol normally begins with a device broadcasting a RARP request, and then waiting for the response (similar to DHCP). However, in the IPnet code, the function *ipnet_eth_rarp_input* is called for every ethernet packet that has the protocol id 0x8035 (The RARP protocol ID):

```
  ...
  if ( *(_WORD *)rarp_pkt == 256
    && *(_WORD *)(rarp_pkt + 2) == 8
    && *(_BYTE *)(rarp_pkt + 4) == 6
    && *(_BYTE *)(rarp_pkt + 5) == 4
    && *(_WORD *)(rarp_pkt + 6) == 1024 )
  {
    interface = ipnet_eth_is_valid_node_mac(interface, rarp_pkt + 8);
    if ( interface )
    {
      // Validate the IP address isn't class D or E
      if ( !(*(_WORD *)(rarp_pkt + 24) & 0x80)
        || (*(_WORD *)(rarp_pkt + 24) & 0xC0) == 128
        || (interface = *(_WORD *)(rarp_pkt + 24) & 0xE0, interface == 192) )
      {
        ret = ipnet_ip4_add_addr(interface);
...
```

Decompilation snippet from *ipnet_eth_rarp_input*

The validations done at the start of this function merely check the validity of the fields of the received RARP packet. Later the provided IP address is simply added by *ipnet_ip4_add_addr*. Additionally, while the address being added is checked to be a class other than D or E addresses, it is not verified that the address is not the local subnet broadcast address, 127.0.0.1, or other highly invalid values. Lastly, There is no limit on the amount of IP addresses that could be added.

This could be used as DoS, by configuring the target device with multiple IP addresses, so that each of those conflicts with other devices on the network, or create invalid routing tables on target device that will prevent it from creating any network traffic.

Logical flaw in IPv4 assignment by the ipdhcpc DHCP client (CVE-2019-12264)

Similar to the RARP vulnerability, an assignment of invalid IPv4 addresses can be achieved by abusing VxWorks' built-in DHCP client (*ipdhcpc*). In all VxWorks' versions that support IPnet (v6.5 and up), the DHCP client will accept any IPv4 address assigned to it by a DHCP server, even if this address is not a valid unicast address (multicast, broadcast, or other illegal addresses).

The function *ipdhcpc_ack_input* will be called when DHCP ACK packets are received, and is responsible for assigning the allocated IPv4 address to the interface. The code in *ipdhcpc_ack_input* will simply take the IP address from the incoming DHCP ACK packet and passes it to the *IP_SIOCAIFADDR* ioctl function that will set it on the interface.

The ability to assign a multicast address to a device, remotely, can lead to problematic scenarios, as will be demonstrated in the following sections.

DoS via NULL dereference in IGMP parsing (CVE-2019-12259)

If an attacker is able to force the assignment of a multicast IP address on a target device via the issue described in the previous section - a NULL dereference that leads to a crash of the network task (*tNet0*) may be achieved through an IGMPv3 membership query packet sent to a target device. This vulnerability affects all VxWorks' versions that support IPnet (v6.5 and up).

When a multicast address is assigned to a target device via the DHCP client the multicast address object in the network interface is not initialized properly. In this state, an attacker can send an IGMPv3 membership query packet to the target device that would lead to a NULL dereference in the function *ipnet_igmpv3_create_membership_report*. In most applications, this would result in a crash of the network task (*tNet0*) since the address 0 will **not** necessarily be mapped, and a page fault would occur.

To trigger this vulnerability an attacker will first force an assignment of a multicast address on a target device via a specially crafted DHCP response packet. Then, he would send an IGMPv3 membership query packet to the target device which will be processed by *ipnet_igmp_input* and an IGMP report will then be scheduled to be sent via the function *ipnet_igmp_report_specific_query*. This function calls the *ipnet_igmp_report* that will attempt to build an *IPNET_MCAST_REPORT_SPECIFIC_QUERY* report:

```
...
v11 = addr_entry->mcast.report_type;
...
if (v11 == 2) {
  v7.set = addr_entry->mcast.filter;
  v7.group_record->record_type = v7.set->user;
...
```

In this specific case, the variable *v7.group_record->record_type* will be set to *addr_entry->mcast.filter->user*. However, in this specific flow, the *filter* pointer inside the *mcast* object will remain set to 0 - so the final dereference (*filter->user)* will result in a NULL dereference. All other cases of the switch case in the above function first validate that the specific set is not NULL before dereferencing it.

When the address 0 is not mapped, this NULL dereference will cause a page fault which would lead to the crash of the network stack (*tNet0*) and result in a DoS for all network related operations of the device.

IGMP Information leak via IGMPv3 specific membership report (CVE-2019-12265)

A general observation we made led us to this vulnerability which can result in an information leak of the packet heap via an IGMPv3 membership query report. The basis of this issue is the introduction of scattered packets in the reassembly of fragmented IP packets. In VxWorks' versions 6.9.3 and up, the reassembly of IP fragments is carried out by linking the various fragments via the *next_part* pointer in the *Ipcom_pkt* structure. This change can be viewed, for example, in the implementation of the *ipcom_pkt_get_length* function:

```
int ipcom_pkt_get_length(pkt_object_t *pkt)
{
    if (pkt->next_part == 0)
        return pkt->end - pkt->start;
    return (pkt->end - pkt->start) + ipcom_pkt_get_length(pkt->next_part);
}
```

The function iterates over all fragments of the packet, and returns their summed length. It seems, however, that not all parts of the IP stack have adjusted for this change, and some functions aren't aware that an IP packet might be scattered. For example, the utility function *ipcom_pkt_get_data*:

```
void* ipcom_pkt_get_data(pkt_object_t *pkt, int offset)
{
    return &pkt->data[pkt->start + offset];
}
```

The *offset* passed to this function might point beyond the first fragment of the IP packet, in one of the additional fragments of the packet, chained to it via the *next_part* pointer. The returned pointer (*&pkt->data[pkt->start + offset]*) might be **outside** the valid range of the first fragment (beyond the *pkt->end*).

An example case of this vulnerability exists In the function *ipnet_igmp_input*:

```
    ...
    pkt_length = ipcom_pkt_get_length(pkt);
    igmp_hdr = ipcom_pkt_get_data(pkt, 0);
    ...
```

The *pkt_length* variable is set using the *ipcom_pkt_get_length* function, that will return the summed length of **all** the packet's fragments, but *igmp_hdr* will point to the first fragment of the packet. The code in this function doesn't validate that the entire *igmp_hdr* or the payload that follows it actually fit in the first fragment.

Later in this function, a set will be built based on the received IPv4 addresses that will follow the IGMP header:

```
    ...
    v14 = ntohs(igmp_hdr->data.igmp.number_of_sources)
    for (v15 = 0; v15 < v14; v15++)
    {
        v32  = ipcom_set_add(sources, &igmp_hdr->data.igmp.source_addr[v15 * 2]);
        ...
```

Snippet from *ipnet_igmp_input*

Since the *igmp_hdr* variable points to the first fragment of the packet, the IPv4 addresses that are being added to the set might actually be coming from an out-of-bound read of that first fragment. The function **does** validate that *pkt_length* contains the needed bytes to avoid parsing out-of-bounds bytes from the incoming packet, but since *pkt_length* is calculated using the entire fragments, this validation is not sufficient.

Continuing in the code-flow of this function we can see that the set containing IPv4 addresses that may have been read out of bounds can be returned to the attacker via the *ipnet_igmpv3_create_membership_report* function that will send an IGMP packet containing this specific set.

A device will be affected by this information leak vulnerability if it has a multicast address assigned to its network interface (any multicast address that is **not** the *all multicast host* address - 224.0.0.1). Using the DHCP client vulnerability described above, an attacker can also force the assignment of a multicast address on a targeted device remotely. However, since this flow will lack some initialization routines that are normally used when a multicast address is assigned the NULL dereference will occur. If the address 0 is a valid, mapped address in the target device, the NULL dereference will **not** crash the network stack, and the information leak would then occur.

Mitigating the risks of URGENT/11 vulnerabilities

Mitigating the risks of the vulnerabilities described in this paper is not a trivial task. Unlike OSs that are used by consumer devices such as PCs and mobile phones, the underlying operating system used by most embedded devices is not regularly advertised. To mitigate the risk of these vulnerabilities, one would first need to identify what devices run VxWorks?

In addition to the difficulty in identifying which devices run VxWorks, device manufacturers are also faced with a challenge to provide firmware upgrades within a reasonable time. Many VxWorks devices, such as medical and industrial devices, are required to go through extensive testing and certification processes before firmware updates can be provided to end-users. Until such updates have been provided, how can end-users protect themselves?

Luckily, there are some unique identifiers for the discovered vulnerabilities that can be used by Firewalls and IDS solutions to detect and block any exploitation attempts of these vulnerabilities.

For example, four of the most critical vulnerabilities we have discovered (CVE-2019-1255, CVE-2019-1260, CVE-2019-1261, CVE-2019-1263) use TCP's Urgent Flag to abuse the Urgent Pointer mechanism of TCP. This mechanism is so remote and unused, that creating rules to detect and block any use of it can be a sufficient method to detect any attempts to attack a VxWorks device with these vulnerabilities.

Firewall rules to drop any TCP packet that has the Urgent Flag turned on can completely eliminate the risk of these 4 vulnerabilities, from attackers coming from the Internet. In addition, IDS solutions can be used to detect attacks in internal networks by detecting **any** use of the Urgent Flag, such as the detection done by following Snort rule:

```
alert tcp any any -> any any (flags: U+; msg: "OS-VXWORKS - Use of Urgent Flag might
indicate potential attempt to exploit an Urgent11 RCE vulnerability";
classtype:attempted-admin; reference:cve,2019-12255; reference:cve,2019-12260;
reference:cve,2019-12261; reference:cve,2019-12263; reference:url,armis.com/urgent11; rev:
1; sid:1000002)
```

Snort rule to detect any use of Urgent Pointer

The above rule can cause some false positives, in the rare case when a legitimate TCP connection uses the Urgent Pointer (such as RLOGIN connections, or certain TELNET clients). An alternative approach can be to limit the detection of the Urgent Pointer vulnerabilities resulting from various state-confusions. These require the use of packets that contain both SYN, URG and FIN flags. This combination will **never** occur in legitimate TCP traffic, and the following Snort rule can detect it:

```
alert tcp any any -> any any (flags: SUF+; msg: "OS-VXWORKS Illegal use of Urgent pointer -
Potential attempt to exploit an Urgent11 RCE vulnerability"; classtype:attempted-admin;
reference:cve,2019-12255; reference:cve,2019-12260; reference:cve,2019-12261;
reference:cve,2019-12263; reference:url,armis.com/urgent11; rev: 1; sid:1000001)
```

Snort rule to detect use of SYN|URG|FIN packets

To detect and block attempts to exploit the IP options vulnerability we've discovered (CVE-2019-12256) one can search for **any** IP packet that contains the LSRR or SSRR options. These options should never be used in modern networks, regardless of the potential RCE vulnerability they present to VxWorks devices. Most firewalls will drop any IP packet that contain these packets for security reasons, and IDS solutions can detect any use of them using the following Snort rules:

```
alert ip any any -> any any (ipopts: lsrr; msg: "OS-VXWORKS Use of LSRR option, potential
attempt to exploit an Urgent11 RCE vulnerability"; reference:cve,2019-12256;
classtype:attempted-admin; reference:url,armis.com/urgent11; rev: 1; sid:1000003)

alert ip any any -> any any (ipopts: ssrr; msg: "OS-VXWORKS Use of SSRR option, potential
attempt to exploit an Urgent11 RCE vulnerability"; reference:cve,2019-12256;
classtype:attempted-admin; reference:url,armis.com/urgent11; rev: 1; sid:1000004)
```

Snort rules to detect any use of LSRR or SSRR options

Conclusion

This research demonstrates some unique vulnerabilities which combine a frightening set of traits: Remotely executable vulnerabilities that don’t require any user interaction, affecting a widely used operating system that drives mission critical devices. In addition, these vulnerabilities allow for some very unique attack scenarios. The IP option vulnerability (CVE-2019-12256) can be used in a broadcast packet and hit any vulnerable device in the local LAN at once. An attacker can use this vulnerability without even being required to take any reconnaissance steps to **find** vulnerable devices within a network. And the TCP Urgent vulnerabilities (CVE-2019-12255, CVE-2019-1260, CVE-2019-1261, CVE-2019-1263) can even be leveraged by an attacker when the target device is located behind a NAT and Firewall solutions - which usually provide impregnable security for the devices behind it.

Although VxWorks includes some optional mitigations that could make some of the URGENT/11 vulnerabilities harder to exploit, we have not found these mitigations used by device manufacturers. In the devices we've examined (and exploited), almost no mitigations were used: no ASLR, no stack canaries and no DEP. Unfortunately, the lack of mitigations makes URGENT/11 vulnerabilities relatively easy to exploit.

The frightening set of traits and attack scenarios these vulnerabilities enable emphasize that RTOSs should receive much more attention and scrutiny by researchers. The challenge of researching closed-source RTOSs should not deter researchers from digging in and continue to uncover vulnerabilities that in some cases can affect critical systems for more than a decade.

20190729.1