ARMIS

Breaking the Discovery Protocols of the Enterprise of Things

Barak Hadad
Ben Seri
Yuval Sarel

Introduction

Armis labs discovered 5 zero day vulnerabilities affecting a wide array of Cisco products, including Cisco routers, switches, IP Phones and IP cameras. Four of the vulnerabilities enable Remote Code Execution (RCE). The latter is a Denial of Service (DoS) vulnerability that can halt the operation of entire networks.

As a group, CDPwn affects a wide variety of devices with at least one RCE vulnerability affecting each device type. By exploiting CDPwn, an attacker can take over organizations' network (switches and routers), it's telecommunication (IP Phones) and even compromise it's physical security (IP Cameras).

Dubbed CDPwn the vulnerabilities reside in the processing of CDP (Cisco Discovery Protocol) packets, impacting firmware versions released in the last 10 years and are an example of the fragility of a network's security posture when confronted with vulnerabilities in proprietary Layer 2 protocols.

The 5 vulnerabilities found are comprised of 4 remote code execution vulnerabilities:

1. Cisco NX-OS Stack Overflow in the Power Request TLV (CVE-2020-3119)
2. Cisco IOS XR Format String vulnerability in multiple TLVs (CVE-2020-3118)
3. Cisco IP Phones Stack Overflow in PortID TLV (CVE-2020-3111)
4. Cisco IP Cameras Heap Overflow in DeviceID TLV (CVE-2020-3110)

And 1 Denial of Service vulnerability:

5. Cisco FXOS, IOS XR and NX-OS Resource Exhaustion in the Addresses TLV (CVE-2020-3120)

This document will detail the attack surface exposed by proprietary Layer 2 protocols as well as the discovered vulnerabilities in the CDP protocol. It will also detail the severe impact these vulnerabilities have if exploited on affected devices.

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:

- URGENT/11 - 11 Zero Day vulnerabilities impacting VxWorks, the most widely used Real Time Operating System (RTOS). The technical whitepaper for this research can be found here:
    - URGENT/11 - Critical vulnerabilities to remotely compromise VxWorks
- 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
- 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

Researching the sub-layers of the network

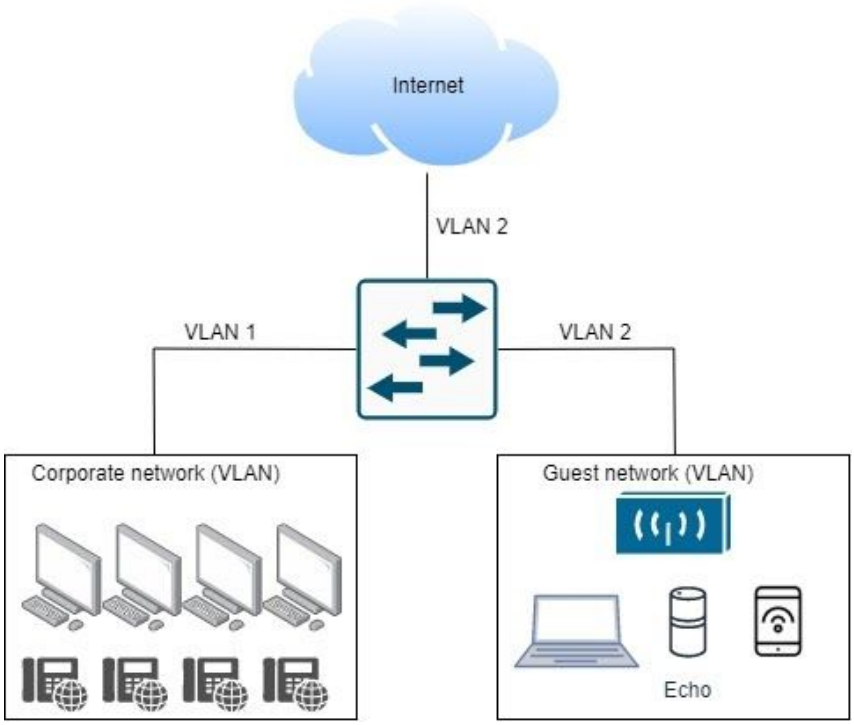
A network of any organization is comprised first by the endpoints that connect to it, but it also contains the devices that operate as the backbone of the network, the pipelines through which traffic is traversed - network appliances such as switches and routers. More often than not, these devices are not taken into consideration when examining the security posture of an organization even though they implement the isolation to sub-networks (a.k.a network segments - VLANs), that is the first line of defense against an attacker that is attempting to perform lateral movement, and move not only between endpoints, but also between segmented parts of the network.

In this research we wish to shed light over the protocols and devices that operate the mechanics of the network, and the vulnerabilities that can arise in their implementations.

What makes a network tick?

A managed network should contain multiple VLANs, split by their level of trust. For example, segregating corporate devices from the Guest WiFi network or from the network containing IoT devices.

The VLANs are handled by an embedded device, without inherent security to it - an enterprise network switch. By taking control of a switch an attacker can traverse between VLANs, breaking the trust zones and gaining access to valuable data, or move laterally to attack additional endpoints. By abusing a relatively vulnerable IoT device located on designated VLAN an attacker could take control of the nearby switch and jump to a mission critical VLAN of the organization.

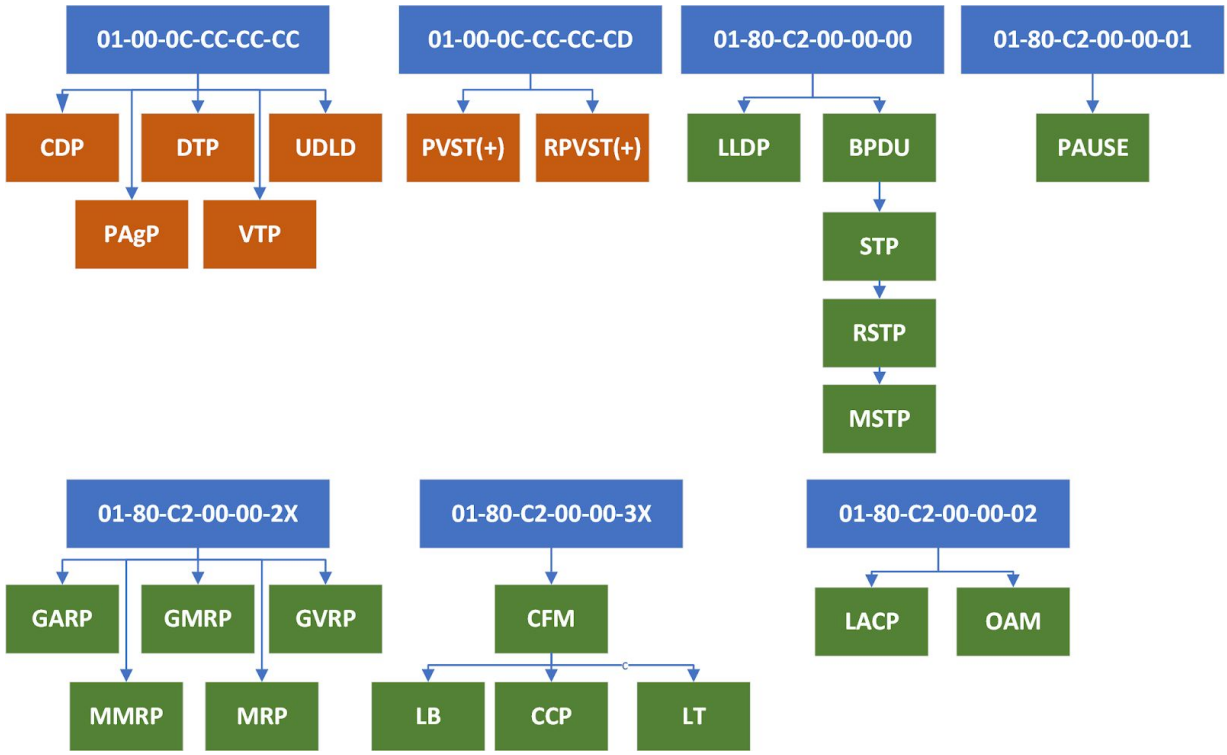


Layer 2 attack surface in network appliances

To accommodate both performance and security needs, network appliances have become increasingly complex implementing dozens of Layer 2 protocols to meet these requirements. These protocols allow many features that were not possible in the past, when networks were powered by simple switches or hubs. The features include network flexibility, “smart” or automatic configuration of ports and efficient network utilization.

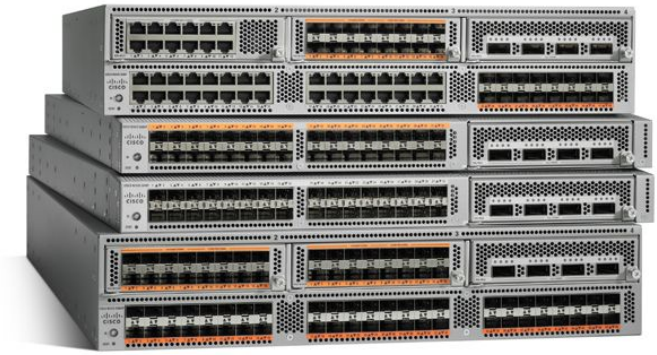
Some of the common protocols that implement these include STP, RSTP and LLDP. Many of the more advanced features of these protocols were created by Cisco - a market leader in the field of network infrastructure. Thus, the protocols are proprietary and not necessarily publicly documented and are supported only by Cisco products. These include CDP, ISL and PVST.

Although these features allow better management and network utilization, they also represent an untapped attack surface that may contain vulnerabilities that allow attackers to take over network appliances and bypass network segmentation.

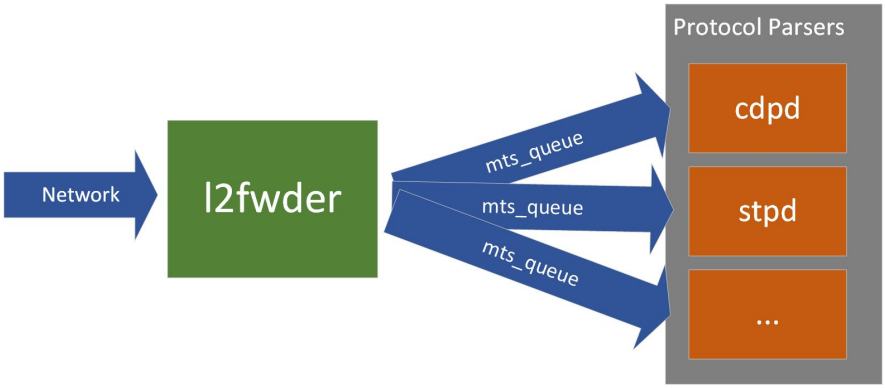


Partial table of Layer 2 protocols used by network appliances, keyed by destination MAC address

While examining a Cisco Nexus switch, for example, we found that many of these Layer 2 protocols are eventually handled by software. Nexus switches are based on the NX-OS operating system which is based on Wind River Linux.



The first step is to find which protocols are routed to the main CPU (the same CPU that parses user input and controls the switch configuration). After a bit of digging we found this data flow:



- Layer 2 forwarder (*l2fwder*) - The first process to parse incoming layer 2 packets, based on some hardware filtering of specific destination MAC addresses. It removes basic packet encapsulation and decides if the packet should be dropped or sent for more specific processing.
- *mts_queue* - A message queue used to send the packets between the processes.
- Protocol parser - Each protocol has its own protocol parser process. For example, the parser for CDP is cdpd.
- *l2fwder* supports a multitude of protocols:
    - BPDU (Bridge Protocol Data Units)
        - STP (Spanning Tree Protocol) - A network protocol that builds a loop free logical topology for Ethernet networks.
            - RSTP (Rapid Spanning Tree Protocol) - Faster STP
            - MSTP (Multiple Spanning Tree Protocol) - STP with VLANs support
    - LACP (Link Aggregation Control Protocol) - Allows port trunking
    - PVST+ (Per VLAN Spanning Tree)
        - RPVST+ (Rapid Per VLAN Spanning Tree)
    - CDP - Cisco Discovery Protocol
    - LLDP - Link Layer Discovery Protocol

- DTP - Dynamic Trunking Protocol
- VTP - VLAN Trunking Protocol

Layer 2 packets of the above protocols are captured on all interfaces of the switch, and not just on a switches management port, and in turn these packets are also parsed by the *l2fwder* process, and the specific protocols parsers as well. By obtaining a vulnerability that can lead to RCE, in any one of the parsing processes of these protocols can allow an attacker to take over a switch, regardless of the VLAN he is in.

Discovery protocols

For a network administrator, it's important to understand what devices are connected to a specific LAN. For this purpose, discovery protocols were invented. They work using periodic advertising packets sent by the connected devices and stored on the switch.

Two discovery protocols are commonly used:

- LLDP - Link layer discovery protocol. Widely used by network printers and other non-Cisco devices.
- CDP - Cisco discovery protocol - Cisco's version of LLDP, enabled by default for almost all Cisco products, including Cisco network appliances, Cisco VoIP Phones, Cisco IP Cameras, etc. Some Cisco features, such as VoIP dedicated VLAN settings rely on CDP and thus it can't be turned off in these devices.

Both protocols are Layer 2 discovery protocols, used by a variety of embedded devices. For large organizations, we can expect both protocols to be enabled. Both of these protocols are based on very flexible structures that enable passing of varying-length fields in multiple formats (TLVs). The various fields and structures in these protocols represent a wide attack surface that might contain vulnerabilities.

Unravelling 1-days in discovery protocols

To better understand this attack surface we searched for any related CVEs disclosed in these discovery protocols, and found a few potential RCEs (remote code execution) in the implementation of CDP and LLDP affecting multiple Cisco products (IOS, IOS XE, IOS XR and NX-OS) that were disclosed in Cisco security advisories. These vulnerabilities were found internally by Cisco, and so these advisories contained almost no description about what exactly was patched. Reverse engineering the patched versions, and comparing them to earlier versions led us to uncover the 1-day bugs that were fixed.

Finding CVE-2018-0167 (LLDP)

The first advisory we've examined describes "Two vulnerabilities in the LLDP subsystem of Cisco IOS Software, Cisco IOS XE Software, and Cisco IOS XR Software could allow an unauthenticated, adjacent attacker to cause a DoS condition or execute arbitrary code with elevated privileges.", the attached bug-report added only a bit more detail:

The vulnerability exists due to improper error handling of malformed LLDP messages. An attacker that is directly connected to an interface of the affected device could exploit this vulnerability by submitting an LLDP protocol data unit (PDU) that is designed to trigger the issue. If successful, an exploitable buffer overflow condition may occur that could result in a DoS condition or the attacker gaining the ability to execute arbitrary code with elevated privileges.

This description is obviously too generic to pinpoint the actual vulnerabilities that were patched. So in search for the bug, we decided to look for the code changes in the various implementation of LLDP in the affected OSes, starting with IOS-XR - A variant of Cisco's widely used IOS operating system used in carrier-grade routers such as the CRS series, 12000 series and ASR 9000 series.

We downloaded the version before the CVE fix (IOS-XRv-5.1.2) and the version right after (IOS-XRv-5.1.3) and compared the two.

Patch-diffing the LLDP daemon versions

We used Diaphora ("diffing plugin for IDA") to compare the two LLDP daemon versions, before and after the fix and found quite a few functions that have changed.

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00036	04208dee	lldp_intf_get_long_name	042058dc	lldp_shim_get_chassis_mac	0.995	4	4	Same rare constant
00000	04200bd2	lldp_timer_handler	04200bd2	lldp_timer_handler	0.990	92	88	Perfect match, same name
00011	042096f2	lldp_if_oper_state_change_cb	042092a4	lldp_if_oper_state_change_cb	0.990	45	43	Perfect match, same name
00017	0421430c	lldp_flood_updates	04213744	lldp_flood_updates	0.980	22	20	Perfect match, same name
00002	04203770	lldp_global_initiation_tasks	0420366e	lldp_global_initiation_tasks	0.970	36	38	Perfect match, same name
00026	0421ab6a	lldp_edm_get_ne_mib_info	04219c4e	lldp_edm_get_ne_mib_info	0.970	1	1	Perfect match, same name
00001	04202dd8	lldp_intf_iteration	04202d5c	lldp_intf_iteration	0.950	76	73	Perfect match, same name
00015	0421008c	lldp_enqueue_unknown_tlv	0420f54c	lldp_enqueue_unknown_tlv	0.950	17	16	Perfect match, same name
00024	0421a0ce	lldp_edm_get_local_addrs	04218ff4	lldp_edm_get_local_addrs	0.940	25	27	Perfect match, same name
00016	04210554	lldp_enqueue_org_def_tlv	0420fa00	lldp_enqueue_org_def_tlv	0.930	20	19	Perfect match, same name
00018	042146cc	lldp_shim_pak_send	04213ac8	lldp_shim_pak_send	0.930	14	12	Perfect match, same name
00012	0420a4b6	is_lldp_supported_on_intf	04209b6e	is_lldp_supported_on_intf	0.920	27	23	Perfect match, same name
00010	04208eee	lldp_intf_cache_info	04208b82	lldp_intf_cache_info	0.900	37	33	Perfect match, same name
00005	04206c68	lldp_shim_update_lldp_mgr	04206a98	lldp_shim_update_lldp_mgr	0.890	26	25	Perfect match, same name
00009	04207e78	lldp_shim_get_global_addr_intf_list	04207c9e	lldp_shim_get_global_addr_intf_list	0.890	32	32	Perfect match, same name
00020	04215122	lldp_shim_packet_init	0421449a	lldp_shim_packet_init	0.870	8	7	Perfect match, same name
00028	0421cc94	sub_421CC94	0421bd50	sub_421BD50	0.860	11	11	Same constants
00007	04207b42	lldp_shim_filter_addr_change_notif	04207a50	lldp_shim_filter_addr_change_notif	0.850	14	20	Perfect match, same name
00014	0420f278	lldp_construct_ma_ipv4_tlv	0420e75e	lldp_construct_ma_ipv4_tlv	0.850	15	12	Perfect match, same name
00030	0421d8b6	sub_421D8B6	0421c94e	sub_421C94E	0.850	10	8	Same constants
00031	0421db86	sub_421DB86	0421cc0a	sub_421CC0A	0.850	10	8	Same constants
00032	0421da1e	sub_421DA1E	0421caac	sub_421CAAC	0.850	10	8	Same constants
00019	04214f0e	lldp_spio_resync_cb	042142d2	lldp_spio_resync_cb	0.840	22	14	Perfect match, same name
00003	04205f32	lldp_shim_get_portnum	04205e08	lldp_shim_get_portnum	0.820	16	14	Perfect match, same name
00022	04215408	lldp_caps_free_router_tlvs	04214704	lldp_caps_free_router_tlvs	0.810	1	1	Perfect match, same name
00035	0421f3a4	sub_421F3A4	0421e340	sub_421E340	0.810	15	13	Same constants
00004	0420612c	lldp_shim_sync_if_config	04205ff2	lldp_shim_sync_if_config	0.800	21	17	Perfect match, same name
00021	04215214	lldp_shim_pak_init	04214588	lldp_shim_pak_init	0.800	12	8	Perfect match, same name
00029	0421d74e	sub_421D74E	0421c7f0	sub_421C7F0	0.800	10	8	Same constants
00013	0420cc28	lldp_process_tlv	0420c220	lldp_process_tlv	0.790	85	66	Perfect match, same name
00033	0421d2de	sub_421D2DE	0421c38e	sub_421C38E	0.760	8	5	Same constants
00025	0421a772	lldp_edm_get_ne_info	042197ea	lldp_edm_get_ne_info	0.740	14	14	Perfect match, same name
00027	0421b15e	lldp_edm_ne_iter_list	0421a24a	lldp_edm_ne_iter_list	0.720	33	32	Perfect match, same name
00034	0421f18c	sub_421F18C	0421e13e	sub_421E13E	0.700	12	12	Same constants
00023	04219a3c	lldp_edm_get_local_addr_list	0421863e	lldp_edm_get_local_addr_list	0.640	63	84	Perfect match, same name
00037	04219586	lldp_edm_insert_mgmt_addr	0420802a	lldp_shim_sort_global_addr_intf_list	0.580	33	50	Same rare constant
00006	042079f0	lldp_shim_cache_one_mgmt_intf_handle	0420786c	lldp_shim_cache_one_mgmt_intf_handle	0.530	19	12	Perfect match, same name
00008	04207cf6	lldp_shim_cache_mgmt_intf_handles	04207c40	lldp_shim_cache_mgmt_intf_handles	0.390	10	5	Perfect match, same name

Screenshot of the *Diaphora* IDA plugin output, comparing two adjacent versions of the LLDP daemon in IOS XR

Most of the changes aboves had nothing to do with the vulnerabilities patched and we needed to find the changes related to the patched vulnerability.

The main TLV parsing function is *lldp_process_tlv* (the added code is marked in green):

```
signed int lldp_process_tlv(int64 a1, pkt_data_t *a_pkt_data, output_t *output,
                            char a4, char *a5, uint64 *a_bytes_read,
                            uint64 a_bytes_left_in_pkt) {
  ...
  switch ( tlv_type )
  {
    ...
    case SYSTEM_CAPABILITIES:
      if ( tlv_length > 255)
        break;
+     if ( tlv_length < 4 )
+       break;
      memcpy(&output->capabilities, &a_pkt_data->tlv_value, 4u);
      ..
    case MANAGEMENT_ADDRESS:
+     if ( tlv_length < 9 )
+       break;
+     if ( output->management_addr_list_len >= 10 )
+       break;
      ma_current_node = output->management_addr_list_head;
```

```
      if ( output->management_addr_list_len > 0 )
      {
        for(index = 0;index < output->management_addr_list_len; index++)
          ma_current_node = (struct_ma_current_node *)ma_current_node->next;
      }
      memcpy(&ma_current_node->addr_str_length, &a_pkt_data->tlv_value, 1u);
      addr_str_length = ma_current_node->addr_str_length;
      bytes_left = tlv_length - 1;
+     if ( addr_str_length <= 1 )
+       break;
      if ( addr_str_length > 32 )
        break;
+     if ( bytes_left < addr_str_length )
+       break;
      memcpy(&ma_current_node->addr_subtype, &a_pkt_data->addr_subtype, 1u);
      memcpy(ma_current_node->management_addr, &a_pkt_data->management_addr,
             addr_str_length - 1);
      interface_subtype = &a_pkt_data->addr_subtype + addr_str_length;
      bytes_left = (unsigned __int16)(bytes_left - addr_str_length);
+     if ( bytes_left < 5 )
+       break;
      memcpy(&ma_current_node->interface_subtype, interface_subtype, 1u);
      memcpy(&ma_current_node->interface_number, interface_subtype + 1, 4u);
      ..
      break;
    case CUSTOM_TLV:
+     if ( tlv_length < 4 )
+       return 0;
      memcpy(output->unknown_tlv_val, &a_pkt_data->tlv_value, 4u);
      ...
  }
```

Simplified decompiled code of the function *lldp_process_tlv*, with the added code marked in green.

From the above code changes, it is apparent that this new version had resolved multiple cases of missing boundary checks that resulted in various flaws:

- Out-of-bound reads of the input packet buffer, that would result in an information leak from the heap stored to the CDP neighbors table.

  For example:

```
case CUSTOM_TLV:
    // DIFF: New check - buffer overread if tlv_length = 0
    //       a_pkt_data->tlv_value might contain out-of-bounds heap data.
+   if ( tlv_length < 4 )
+       return 0;
    memcpy(output->unknown_tlv_val, &a_pkt_data->tlv_value, 4u);
```

- Integer underflows that would result in a *memcpy* of size *MAX_UINT* (a.k.a “wild-copy”), that would create uncontrolled heap overflows that would ultimately crash the CDP daemon.

For example:

```
case MANAGEMENT_ADDRESS:
    ...
    // DIFF: new check - crash due to huge memcpy
+   if (addr_str_length <= 1)
+       break;
    ...
    memcpy(ma_current_node->management_addr, &a_pkt_data->management_addr,
           addr_str_length - 1);
```

- NULL dereference, due to lack of size validation of certain linked-lists. In most cases this would also be an unexploitable flaw, and will lead to a crash of the CDP daemon.

  For example:

```
case MANAGEMENT_ADDRESS:
  ...
  management_addr_list_len = output->management_addr_list_len;
  // DIFF: new check - Crash caused by writing to NULL (end of list)
+ if ( management_addr_list_len >= 10 )
+   break;
  // List size is hard coded to 10!
  ma_current_node = output->management_addr_list_head;
  if (management_addr_list_len > 0)
  {
    for (index = 0;index < management_addr_list_len; index++)
      ma_current_node = (struct_ma_current_node *)ma_current_node->next;
  }
  memcpy(&ma_current_node->addr_str_length, &a_pkt_data->tlv_value, 1u);
```

These vulnerabilities could be easily used for DoS attacks. However, no exploitable RCE vulnerability was found in these code changes, and we decided to move on to the 1-days disclosed in the CDP protocol.

Finding CVE-2018-0303 (NX-OS and FXOS RCE in CDP)

From all the security advisories published by Cisco regarding vulnerabilities in layer-2 discovery protocols, only one clearly indicated that an RCE vulnerability was patched. This advisory focused on the NX-OS and FXOS operating systems, and detailed a buffer-overflow vulnerability in parsing of CDP (Cisco Discovery Protocol) packets:

> *A vulnerability in the Cisco Discovery Protocol component of Cisco FXOS Software and Cisco NX-OS Software could allow an unauthenticated, adjacent attacker to execute arbitrary code as root or cause a denial of service (DoS) condition on the affected device. The vulnerability exists because of insufficiently validated Cisco Discovery Protocol packet headers. An attacker could exploit this vulnerability by sending a crafted Cisco Discovery Protocol packet to a Layer 2 adjacent affected device. A successful exploit could allow the attacker to cause a buffer overflow that could allow the attacker to execute arbitrary code as root or cause a DoS condition on the affected device.*

This time, it seemed an exploitable RCE vulnerability might be within reach.

Similarly to the 1-day vulnerability research done in LLDP, we examined the code before and after the patched version and found a heap overflow in *cdpd_handle_addr_tlv*:

```
void *cdpd_handle_addr_tlv(device_record_t *dev_record, uint16 packet_len,
                           addresses *in_packet)
{
  ...
  number_of_addr = in_packet->num_of_addr;
  // same dev_record is used for all CDP packets from the same device
  if (32 * number_of_addr > dev_record->addr_blob_used_bytes)
  {
    dev_record->addr_blob = cdpd_malloc(8, 32 * number_of_addr);
    dev_record->addr_blob_used_bytes = 32 * number_of_addr;
    ...
  }
  addr_blob = (struct_addr_blob *)dev_record->addr_blob;
  ...
  for (in_addr in in_packet->addresses)
  {
    ...
    addr_len = in_addr->addr_len;
    if ((in_packet->protocol_type == IPV4) && (addr_len == 4))
    {
      // protocol_type is not initialized on new packets
      addr_blob->protocol_type = IPV4;
    }
    else if ((in_packet->protocol_type == IPV6) && (addr_len == 16))
    {
      addr_blob->protocol_type = IPV6;
    }
    if ((addr_blob->protocol_type == IPV4) || (addr_blob->protocol_type == IPV6))
    {
      memcpy(&addr_blob->addr, in_addr->addr, addr_len);
    }
    ...
  }
```

Simplified decompiled code of the function *cdpd_handle_addr_tlv*, disassembled from NX-OS v7.3.0.1

The variable *addr_blob* holds the parsed address field values sent from a certain CDP neighbor. This structure is allocated on the heap, each time the size of an incoming addresses TLV array is larger than the buffer currently allocated for a certain CDP neighbor. If the incoming CDP packet contains an addresses TLV array of similar, or smaller size - the *addr_blob* allocated previously is reused.

In addition to this behaviour, this code lacks proper validation of the *addr_len* field. It will validate that an *IPV4* address is 4 bytes long, and that an *IPV6* address is 16 bytes long, and will set the *addr_blog->protocol_type* accordingly. However, since the entire *addr_blog* is reused between instances of incoming CDP packets, the above validation might be insufficient.

The main bug in this function is that *addr_blob→protocol_type* is used for flow control but it isn't reinitialized on new packets - causing a state confusion between calls to this function that can ultimately allow an attacker to freely control the *addr_len* variable passed to *memcpy* at the end of the function which will lead to heap-overflow with attacker controlled data.

Consider the following scenario:

1. A legitimate CDP packet is processed, containing one address with *protocol_type* = 1(IPv4)
    a. *dev_record* is allocated for the specified MAC address
    b. *addr_blob* is allocated for the dev_record and *addr_blob→protocol_type* = IPV4
2. A second CDP packet, is processed from the same CDP neighbor. This time, the packet is malicious, containing one address with *protocol_type* = 0xA (Not *IPV4* or *IPV6*) and *addr_len*=0xffff
    a. The same *dev_record* is used because it's the same MAC address
    b. Since the *number_of_addr* didn't change (still 1), *malloc* is not called again and we get the same *addr_blob* as for the previous packet with *addr_blob→protocol_type = IPV4* already set.
    c. Inside the for loop, *addr_blob→protocol_type* is 0xA so we don't enter the first two if clauses and *addr_blob→protocol_type* still equals *IPV4*.
    d. Since *addr_blob→protocol_type* was not set in this iteration, it equals *IPV4* (from the previous call to the function)
        i. *memcpy* is called with size *addr_len* (attacker controlled) even though the target buffer is of size 32 bytes. This will cause an out-of-bounds copy onto a heap buffer.

Using some heap shaping this vulnerability could lead to remote-code-execution.

Continuing the research from the above flow, we discovered a zero-day DoS vulnerability (CVE-2020-3120) in the processing of the same TLV presented above.

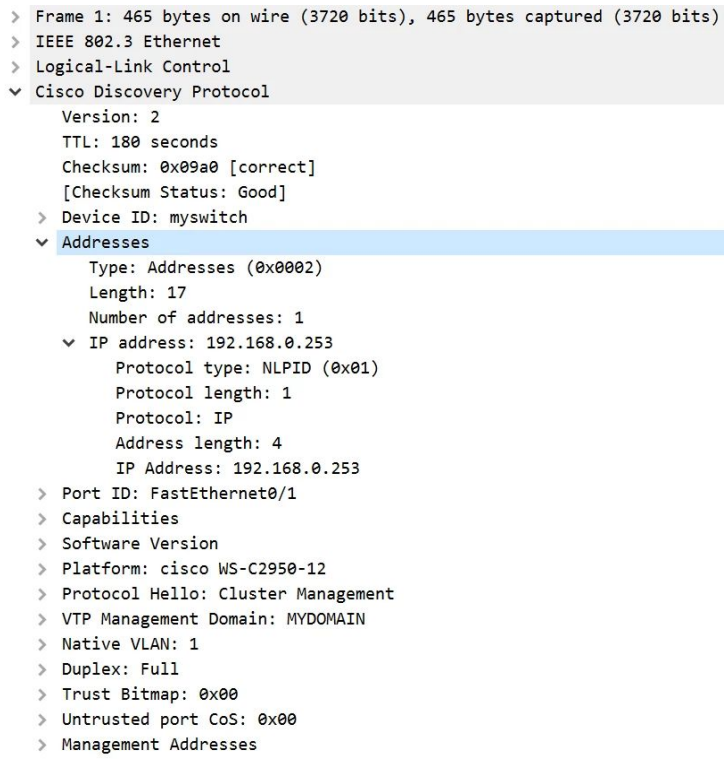
Newly discovered vulnerabilities in CDP

Having learned the common vulnerabilities in Cisco discovery protocols parsers, we set out to find zero-day vulnerabilities that were not patched at the time.

CDP vulnerabilities in switches and routers

NX-OS Resource Exhaustion in the Addresses TLV (CVE-2020-3120)

Examining the structure of the Addresses TLV in depth, raises some concern on the difficulty of parsing it bug-free:



There are 4 varying length fields in the same TLV:

- TLV Length - 2 bytes
    - Number of addresses - 4 Bytes (Why have a 4 byte length that is encapsulated by a 2-byte length? 😐)
        - Protocol length - 1 Byte
        - Address length - 2 Bytes

Some of these length fields are also redundant, when considering they represent the size of IP addresses, while the type of the IP address is also part of the TLV structure (an IPv4 address will **always** be 32-bit, for example).

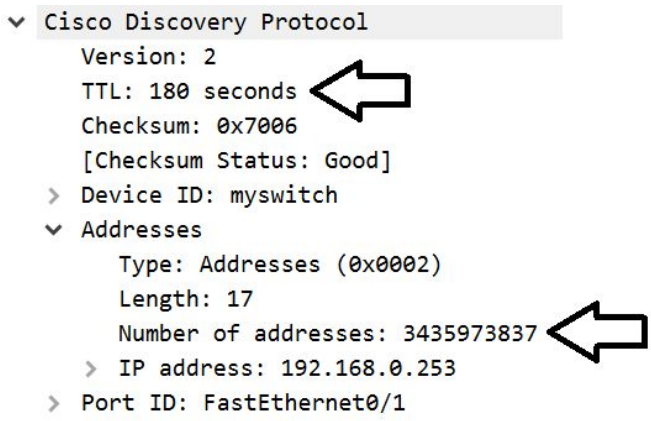
One edge-case problem introduced by this structure, is the ability of an attacker to cause allocations of huge memory blocks by setting *Number of Addresses* to any size he wishes, which can eventually lead to a crash of the CDP daemon.

Let's have another look on the *cdpd_handle_addr_tlv* function (in NX-OS v7.3.0.1):

```
void *cdpd_handle_addr_tlv(struct_device_record *dev_record,
                           uint16 packet_len, addresses *in_packet)
{
  ...
  number_of_addr = in_packet->num_of_addr;
  if (32 * number_of_addr > dev_record->addr_blob_used_bytes)
  {
    cdpd_free(dev_record->addr_blob);
    // This buffer is allocated even if the packet doesn't have enough addresses
    dev_record->addr_blob = cdpd_malloc(8, 32 * number_of_addr);
    dev_record->addr_blob_used_bytes = 32 * number_of_addr;
  }
```

Decompiled code snippet *cdpd_handle_addr_tlv*

Regardless of the fact the CDP packet **is** limited by size, the *number_of_addr* field isn't boundary checked by the above code, and an attacker can cause allocations of huge blocks of memory. The *dev_record->addr_blob* variable is freed only after the CDP record timeout passes, and this timeout is also sent in the CDP packet, allowing the attacker to control when the allocation is freed. This means that an attacker can control exactly how much memory is going to be allocated, and for what period of time, quickly exhausting the device memory.



In the latest release of NX-OS at the time of this research (v9.2.3), and additional validation of *number_of_addr* was introduced:

```
void *cdpd_handle_addr_tlv(device_record_t *dev_record,
                           uint16 packet_len, addresses *in_packet)
{
  ...
  number_of_addr = in_packet->num_of_addr;
  // Overflow check?
  if ( number_of_addr > 32 * number_of_addr)
  {
    ...
    return &n + 1;
  }
  if ( 32 * number_of_addr > dev_record->addr_blob_used_bytes )
  {
    cdpd_free(dev_record->addr_blob);
    dev_record->addr_blob = cdpd_malloc(8, 32 * number_of_addr);
    ...
```

Decompiled code snippet *cdpd_handle_addr_tlv*

To prevent a possible flow, where the allocation size of *number_of_addr * 32* overflows, which in turn might lead to a small buffer allocated, while a large amount of addresses are parsed and copied to that allocation buffer, the overflow check above was introduced. Unfortunately, the above condition can still overflow and this validation can be subverted. Multiplying an integer by 32 is the same as shifting it left by 5 bits, meaning that in order to pass this condition, an attacker just needs to set the most significant 5 bits to a smaller number than the next 5 bits.

To clarify this statement, consider the following 16-bit number:

0b0111110000000000 * 32 = 0b0111110000000000 << 5 = 0b1000000000000000
0b0111110000000000 < 0b1000000000000000

This multiplication can thus overflow, and remain **larger** than the original number. So despite this added validate, an attacker can still allocate huge amounts of memory leading the device to crash.

IOS-XR variant of CVE-2020-3120

The same TLV parsing mechanism was examined in the latest version of IOS-XR (v6.5.2), and a different bug with a similar effect was discovered.

The vulnerable function *cdp_handle_address_info* will also allocate an *addr_blob* with minimal boundary check validations:

```
if (bytes_left_in_packet >= (unsigned int)(5 * number_of_addresses))
{
  ...
  device_entry->addr_blob = malloc(24 * number_of_addresses);
  ...
```

Decompiled code snippet from the function *cdp_handle_address_info*

In this implementation, *number_of_addresses * 5* is validated to be smaller than the number of bytes left in the packet. However, the allocations size following this validation is of *number_of_addresses * 24*. A 32-bit number that will be overflowed to a very small number when multiplied by 5, can simultaneously be overflowed to a large number when multiplied by 24 that will also lead to a large allocation.

Any product of $\frac{MaxUInt}{5}$ will pass the first check so we will use $\frac{4 \times MaxUInt}{5}$ = 0xccccccccd

0xccccccccd * 24 = 0x33333338 = ~820 MB

Exploitability

An attacker could exploit the variants of this vulnerability as described above by sending multiple CDP packets and precisely control the amount of memory he wishes to exhaust by controlling the number of addresses field.

Impact

Using this vulnerability an attacker can crash the CDP process multiple times. On both NX-OS and IOS-XR, the device will reboot after a few CDP daemon crashes, meaning that using this vulnerability, an attacker can cause a complete DoS of the target device.

NX-OS Stack Overflow in the Power Request TLV (CVE-2020-3119)

Following these findings, we further examined the parsing function in the CDP daemon on the latest NX-OS version (v9.2.3) at the time, in search for any TLV types that might contain similar boundary check flaws. Eventually we found the Power Request TLV - a CDP TLV frame made for negotiation of Power-over-Ethernet parameters.

The Power Request TLV contains a list of requested power specifications. The 16-bit list length is not validated correctly and used to copy the list to a fixed size buffer on the stack and a fixed offset from an additional pointer (*a1*) -- also on the stack:

```
int length = ntohs(pwr_pkt_2->length);
...
if (length > 0) {

  Current_offset = &pwr_pkt_2->int8;
  counter = 1;
  do {
    // Overflow - temp is a buffer of size 0x40 at the top of the stack frame
    temp[counter - 1] = _byteswap_ulong(*current_offset);
    ...
    // Write, What, Where primitive since a1 is on the stack
    a1->levels[counter] = _byteswap_ulong(*current_offset);
    ++current_offset;
    ++counter;
  } while (counter != length + 1);
}
```

Decompiled code snippet from the CDP parsing function

An attacker can exploit this vulnerability using aCDP packet with more than 16 power levels:

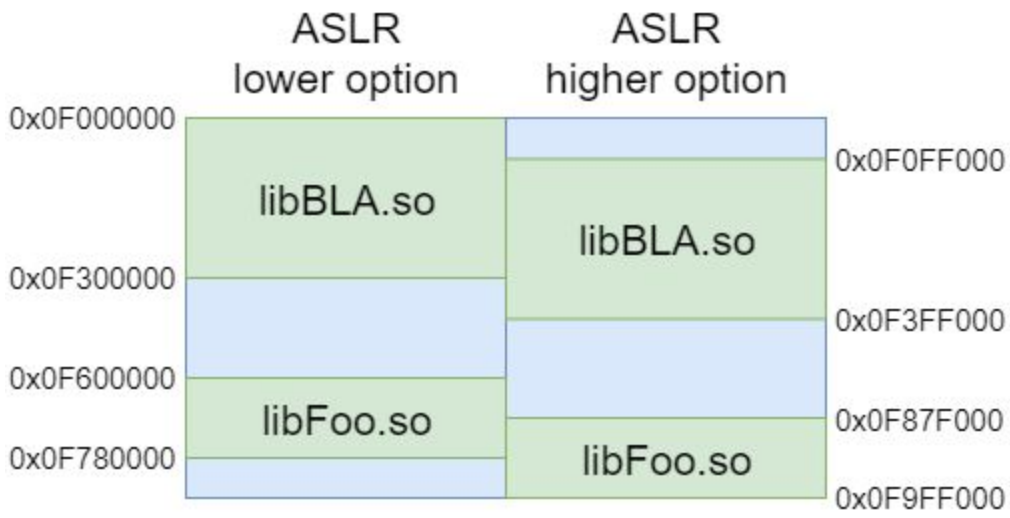
```
> Frame 1: 465 bytes on wire (3720 bits), 465 bytes captured (3720 bits) on interface 0
> IEEE 802.3 Ethernet
> Logical-Link Control
v Cisco Discovery Protocol
    Version: 2
    TTL: 180 seconds
    Checksum: 0x038c [correct]
    [Checksum Status: Good]
  > Device ID: myswitch
  > Addresses
  > Port ID: FastEthernet0/1
  > Capabilities
  v Power Request: 1864386926 mW, 1952805486 mW, 1702131567 mW, 1919623247 mW, 188569660
      Type: Power Requested (0x0019)
      Length: 276
      Request-ID: 17257
      Management-ID: 29539
      Power Requested: 1864386926mW
      Power Requested: 1952805486mW
      Power Requested: 1702131567mW
      Power Requested: 1919623247mW
      Power Requested: 1885696609mW
      Power Requested: 1953066599mW
      Power Requested: 542341491mW
      Power Requested: 1952804128mW
      Power Requested: 1399809652mW
      Power Requested: 2002874981mW
      Power Requested: 537545039mW
      Power Requested: 1394616436mW
      Power Requested: 1831411779mW
      Power Requested: 842609968mW
      Power Requested: 542338918mW
      Power Requested: 1953980786mW
      Power Requested: 1696606275mW
      Power Requested: 842609968mW
      Power Requested: 759772747mW
      Power Requested: 843854417mW
      Power Requested: 875384105mW
```

Exploitability

The above vulnerability is a stack overflow, and the CDP daemon in which it was found is not using the stack canaries mitigation. ASLR, however, is in use by the CDP daemon, but the daemon is a 32 bit process and ASLR is only partially effective. In most cases, ASLR in a 32 bit process in Linux only randomizes one byte of the address. Moreover, the entire memory map is shifted with the same offset and the distance between two adjacent ASLR shifts is just 4KB, making the maximum ASLR shift - 1MB (256 * 4KB).

Any shared object larger than 1MB (and there are a few in use by the CDP daemon), an attacker can choose addresses within these libraries that will necessarily contain executable code that can be used for ROP (Return oriented programming) gadgets, in multiple ASLR shifts.

Using the above technique, we were able to develop a relatively reliable exploit of this vulnerability that is able to gain remote-code-execution within reasonable time.



For example, In the illustration above, address 0x0F100000 is a valid executable address for all possible ASLR shifts. Using the above limitation of ASLR in the CDP daemon, an attacker can develop an exploit that will use a ROP chain that will handle concurrent options for the ASLR shift, and can thus dramatically limit the effect of the ASLR mitigation.

Impact

As described above, a reliable exploit that reaches code-execution within a reasonable amount of time can be achieved. Since the CDP daemon is running with root privileges, such an exploit would allow an attacker full control over targeted switches, which in turn can allow him to hop between VLANs and causing havoc to the network structure.

IOS XR Format String vulnerability in multiple TLVs (CVE-2020-3118)

In the IOS XR implementation of CDP the above TLVs didn't contain similar vulnerabilities. However, in the code flow triggered when certain string-fields are copied from a CDP packet to the in-memory database of the router's CDP neighbors, we found a format-string vulnerability:

```
v5 = (int *)calloc(1, v38);
snprintf((int)(v5 + 7), 0x1E, device_id);
snprintf((int)v5 + 58, 0x28, port_id);
snprintf((int)v5 + 98, 0x20, software_version);
```

Decompiled code snippet from the parsing flow of CDP packets in IOS-XRv

The three fields above - Device ID, Port ID, and Software Versions, are attacker controlled and can be used to overwrite an out-of-bounds stack variable, that can lead to remote-code-execution, for example by using the *%n* modifier. (https://en.wikipedia.org/wiki/Uncontrolled_format_string)

Exploitability

This vulnerability affects a wide-array of IOS XR versions, and substantial underlying OS changes have occured between these versions. In IOS XR v6.1.3, for example, the underlying OS was based on QNX, which didn't support ASLR, making this vulnerability easily exploitable.

In the latest releases of IOS XR, however, the underlying OS is based on Windriver Linux, the CDP process is 64-bit and ASLR is enabled -- meaning exploitation is not as trivial. Nevertheless, an attacker could use this vulnerability to accurately corrupt variables on the stack, by abusing the flexible out-of-bound write primitives enabled by the format-string vulnerability. These can in turn also lead to code execution.

Impact

Using this vulnerability, an attacker could gain full control over the target router, hop between network segments and use the router for subsequent attacks.

CDPwn all the things - CDP vulnerabilities in IP Phones & Cameras

Our initial research focused on the security of network segments, as implemented by network appliances, and thus the Cisco products we've examined were Cisco Nexus switches running NX-OS, and Cisco routers running IOS XR. Having found a new vulnerability in the CDP implementation in NX-OS, we set out to see if the same vulnerability affects the CDP code base used by additional Cisco devices. To our surprise, it seems the CDP code base is separate for each Cisco product line we've examined.

Examining the CDP implementation in Cisco VoIP Phones, and Cisco Video Surveillance products (IP Cameras) led us to discover two RCE zero-day vulnerabilities.

IP Phones Stack Overflow in PortID TLV (CVE-2020-3111)

Cisco VoIP devices use CDP for multiple reasons, one of which is PoE (Power over Ethernet) negotiation - The VoIP device can request specific PoE parameters and the switch can enable or disable those parameters and inform the VoIP using CDP. For this reason, CDP is enabled by default in the Cisco VoIP devices, and can not be easily turned off.

The underlying operating system for Cisco-88xx and 78xx VoIP Phones is Linux and the process running the cdp daemon is executed with root privileges.

Examining the function that parses incoming CDP packets (*cdpRcvParse*) in the VoIP's CDP daemon (cdpd) led us to discover a stack overflow vulnerability in the parsing of the PortID TLV (0x03). There are no boundary checks on the length of this TLV and the value is simply copied to a fixed sized buffer on the stack.

```
case 3: # Port ID
  len = ntohs(portid_tlv_len);
  if ( len > 3 )
  {
    // No check that len is lower than the size of buf (stack variable)
    memcpy(buf, v10 + 4, len - 4);
    ..
```

Decompiled code snippet from *cdpRcvParse*

Exploitability

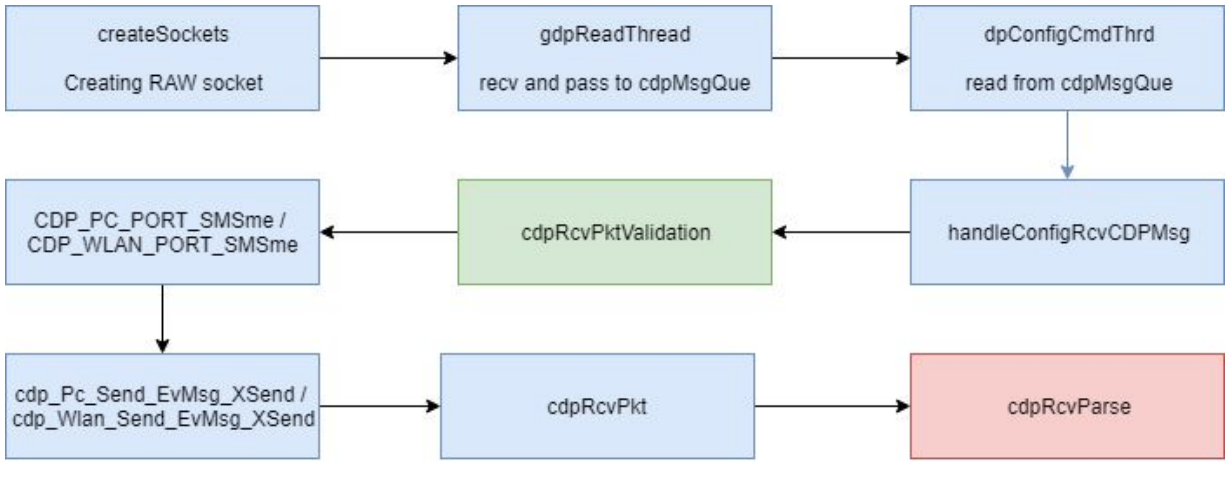
The above vulnerability is a simple stack-overflow, with attacker controlled data. Stack canaries are not in use by the CDP daemon, but ASLR is. However, similarly to the CDP daemon used in the NX-OS, ASLR is only partially effective. Using the same techniques, we were able to develop a relatively reliable exploit of this vulnerability that is able to gain remote-code-execution. We were also able to craft an ethernet broadcast packet causing DoS on all vulnerable devices on the same LAN.

Impact

In a network comprised of CDP-capable network switches (most likely, Cisco switches) CDP packets are terminated by each switch. CDP packets have a designated multicast address to which they are sent (01:00:0C:CC:CC:CC), and each CDP-capable switch captures packets sent to this MAC address, and does not forward them throughout the network. Thus, in a CDP-capable network, an attacker can only trigger the above vulnerability by sending CDP packets when it is directly connected to a target device.

However, an additional flaw was discovered in the parsing mechanism of CDP packets in the VoIP phones, enhancing the impact an attacker can achieve using the vulnerability. The CDP implementation in the VoIP phones doesn't validate the destination MAC address of incoming CDP packets, and accepts CDP packets containing unicast/broadcast destination address as well. Any CDP packet that is sent to a switch that is destined to the designated CDP multicast MAC address, **will** be forwarded by the switch, and not terminated by it. Due to this discrepancy, an attacker can trigger the vulnerability described above by a unicast packet sent directly to target device, or by a broadcast packet sent to all devices in the LAN -- without needing to send the packet directly from the switch to which an VoIP phones is connected to.

To understand this additional bug, we examined the flow of an incoming packet:



Code flow of an incoming CDP packet on the Cisco 88xx and 78xx IP Phones

The majority of the above flow simply sets up a RAW socket to capture the CDP packets, while the validation of the packet occurs in the function *cdpRcvPktValidation*:

```
if ((packet->llc.dsap == 0xAA) &&
    (packet->llc.ssap == 0xAA) &&
    (!memcmp((char *)&packet->llc.control_field + 1, &s2, 3u)) &&
    (LOBYTE(packet->llc.pid) == 32) &&
    (!HIBYTE(packet->llc.pid))) {

    if (!memcmp(packet->eth.srca, myMAC, 6u)) {
        syslog(4, "pktvalid(): pkt dropped: my mac address: \n");
    } else if ((cdp->version == 1) || (cdp->version == 2)) {
        if (cdpChksum((uint16_t *)cdp, payload_size - 22)) {
            syslog(4, "pktvalid(): pkt dropped: checksum error \n");
        }
    ...
    }
}
```

Decompiled code snippet from the function *cdpRcvPktValidation*

The only validation of the ethernet header fields is of the source MAC address, which is validated to be any address that isn't the address of the device itself. The destination address isn't validated at all! This means that packets sent to the VoIP with a unicast or broadcast address will also be processed.

As described above, this flaw increases the impact of this vulnerability, since CDP packets sent with unicast\broadcast destination addresses are not processed by other network devices and will be forwarded in the LAN without being terminated by CDP-capable network appliances. This allows for 2 new attack scenarios:

1. An attacker in the same LAN, without a direct connection to the target device, can attack it via a unicast packet.
2. An attacker can send a broadcast message containing a malicious packet can cause a DoS on all vulnerable devices in the LAN, and potentially reach code execution as well.

IP Cameras Heap Overflow in DeviceID TLV (CVE-2020-3110)

We were surprised to see that Cisco also manufactures IP cameras and that those cameras use yet another code base for CDP processing with it's own brand of CDP vulnerability:

```
case 1u:                        // DEVICE ID
     uint16 value_len = (len - 4) & 0xFFFF;
     uint8  alloc_len = (len + 1) & 0xFF;

     // Cast to uint8
     dst_buf = malloc(alloc_len);
     if ( dst_buf )
     {
       memset(dst_buf, 0, alloc_len);
       // Cast to uint16 - heap overflow
       memcpy(dst_buf, tlv + 4, value_len);
       // Place a NULL terminator at the end of the buffer.
       dst_buf[value_len] = 0;
       ..
       goto LABEL_12;
```

Decompiled code snippet from the function *cdpd_process_packet*

The above code simply allocates a buffer for the Port ID parsed from the incoming packet, and copies it's value from the incoming TLV to the allocated buffer. However, a simple mistake here means a trivial heap-overflow can occur. The size of the allocation dst_buf is calculated as len + 1, to allow the addition of a null terminator at the end of the Port ID string. Unfortunately, this variable (alloc_len) is defined as an uint8, while the size calculated as the length of the TLV's payload (value_len) is defined as an uint16. The length field in the TLV is 16-bit, and is completely attacker-controlled. By sending a CDP packet containing a PortID TLV (0x01) of size larger than 0xff an attacker can overflow a heap allocated buffer with attacker-controlled data.

Exploitability

The above vulnerability is a heap-overflow with attacker controlled data that can lead to remote-code-execution. An attacker sending maliciously crafted CDP packets to a target device can use the various allocations that occur while handling CDP packets to shape the heap prior to triggering the overflow. Moreover, multiple overflow packets can be sent by the attacker with varying overflow lengths. Lastly, the Cisco IP Camera's CDP daemon is a non-position independent process, that is always mapped at a constant address -- meaning ASLR is not in use in this process code section.

Combining the above traits, an experienced attacker can develop a statistical, yet reliable working exploit that may reach RCE.

Impact

Unlike the example of the VoIP Phones, the IP Cameras do validate the ethernet destination address in incoming CDP packets, and only process those that are sent to the dedicated multicast address.

As stated above, in a network comprised of CDP-capable network switches (most likely, Cisco switches) CDP packets are terminated by each switch. In such a network, an attacker can thus only trigger the

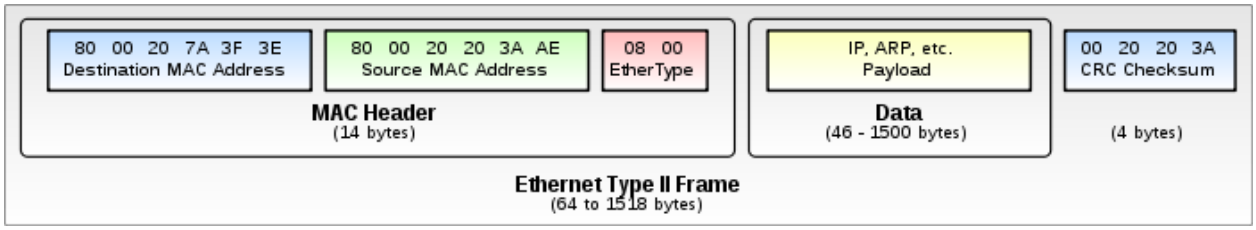
above vulnerability by sending CDP packets when it is directly connected to a target IP camera, which would mean running his attack from the camera's access switch. In a network that isn't comprised of Cisco switches, CDP packets aren't terminated at each switch, and the attacker can then send the multicast CDP packet to any IP camera in the LAN.

Despite the above, there are a few edge cases when seemingly valid multicast CDP packets can still traverse through CDP-capable switches, due to a small discrepancy in how ethernet header is parsed by the IP cameras (that run on Linux) vs how it is parsed by certain network switches (that might be using a non-Linux TCP/IP stack).

In order to capture the CDP packets, the CDP daemon running in the IP camera opens a raw socket, such as this:

```
cdp_socket = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_802_2));
```

The above raw socket will pick up any ethernet packet that is of type *ETH_P_802_2*. Ethernet 802.2 is a logical link control layer (LLC), that expands the basic ethernet header:



The 16-bit EtherType field in the Ethernet header can actually also function as ther Length field of the Ethernet frame:

> *Values of 1500 and below mean that it is used to indicate the size of the payload in octets, while values of 1536 and above indicate that it is used as an EtherType, to indicate which protocol is encapsulated in the payload of the frame.*

Description of the EtherType field, from Wikipedia

When the EtherType is thus less than 1500 bytes, it is treated as the size of the Ethernet payload, which also indicates that an LLC frame will exist above the Ethernet frame. The Linux Kernel will thus capture *ETH_P_802_2* frames (LLC frames) according to these conditions:

```
#define ETH_P_802_3_MIN 0x0600 // (1500)

static inline bool eth_proto_is_802_3(__be16 proto)
{
    ...
    /* cast both to u16 and compare since LSB can be ignored */
    return (__force u16)proto >= (__force u16)htons(ETH_P_802_3_MIN);
}
```

So according to Linux Kernel, any EtherType greater than 1500 will be considered as a LLC packet, while

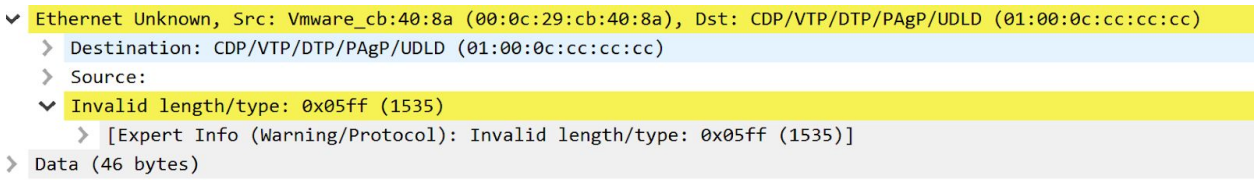
the Ethernet RFC is actually defined a slightly different:

> *...values of 1500 and below for this field indicate that the field is used as the size of the payload of the Ethernet frame while values of 1536 and above indicate that the field is used to represent an EtherType. The interpretation of values 1501–1535, inclusive, is undefined*

Description of the EtherType field, from Wikipedia

So when an Ethernet packet containing EtherType field of 1501-1535 is received by the Linux Kernel, the EtherType will be interpreted as the length of the packet, and assumed to be a valid 802.2 LLC packet, while other network appliances might treat the EtherType as a protocol type, and not assume it contains LLC frame at all.

For example, the packet capturing tool Wireshark does not know how to parse this type a crafted packet containing this type of EtherType field, and simply specifies it contains an invalid EtherType:



If a network appliance where to treat this field as a valid protocol type, it might than simply forward the packet throughout the network, since CDP packets that are usually terminated by CDP-capable network appliances have to include a valid LLC frame within them.

Several tests we've performed showed that Nexus switches, for example, simply drop these types of packets. However, we have not done a thorough tests on all CDP-capable network appliances, and the exact interpretation they have on the EtherType field will determine if this subtle attack can be used to send a malicously crafted CDP packet to affected Cisco IP Cameras, even without needing to be directly connected to them.

Conclusion

Many of the vulnerabilities described in this research are simple overflows, that are becoming less frequent in secure software. However, it seems that in both discovery protocols supported by default on enterprise-grade network appliances (LLDP and CDP), these types of trivial flaws were found extensively.

When simple flaws are discovered, they can ultimately be exploited to reach remote-code-execution, and to carry out attacks on vulnerable devices. The risk to the network appliances that serve as the security guard, preventing compromised devices from crossing over network segments, is great.

Moreover, attacks that leverage CDP vulnerabilities against enterprise-grade IP Phones and cameras can also be substantial. In the case of IP Phones, this research demonstrated one of the most severe outcomes for an RCE vulnerability -- the ability for an attacker to send an unauthenticated broadcast packet that would traverse through the LAN and affect any affected IP Phone it reaches. Gaining code execution en mass in this scenario is not a trivial task, but the possibility of it is real.

In addition to the discovered vulnerabilities, it seems the attack surface of Layer 2 protocols, used by network appliances is significant and largely unexplored. These protocols are in use by a wide array of devices, and are enabled by default in the majority of them.

We hope this research will shed light on this attack surface, and the risks that it entails.