ARMIS

CDPwn

Bypassing Mitigations While Taking Over Enterprise-IoT Devices in Broadcast Attacks

Barak Hadad
Ben Seri

Table of Contents

Introduction

In February of 2020, Armis Labs disclosed the discovery of 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 an organizations' network (switches and routers), its telecommunication (IP Phones), and even compromise its 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.

On October 20, 2020, the NSA published a report identifying the Top 25 vulnerabilities that are currently being consistently scanned, targeted, and exploited by Chinese state-sponsored hacking groups. One of the five RCEs in CDPwn (CVE-2020-3118) was identified on this list, stressing the importance of these vulnerabilities, and the need to mitigate their potential effects.

While the criticality of the discovered RCEs was not in question, the exploitability of them remained something we felt the need to prove. Unlike some embedded devices, Cisco does employ some exploit mitigations in the affected products, such as ASLR (address space layout randomization) and others. Overcoming ASLR, for example, with the RCE memory corruption vulnerabilities found in CDPwn can be quite tricky since CDP is primarily a one-directional protocol, meaning the likelihood of discovering an additional information leak vulnerability that would assist in the bypass can prove challenging.

This document details an ASLR bypass technique that we developed to exploit the CDPwn vulnerabilities successfully without the need for additional information leak vulnerabilities. Since some of the vulnerabilities can be sent over a broadcast packet, a blind ASLR bypass technique was also developed that can allow an attacker to take control over multiple vulnerable devices **simultaneously**.

Although ASLR can be effective mitigation to prevent a memory corruption vulnerability from reaching code-execution, its effectiveness highly depends on the specifics of its implementation. The basic purpose of ASLR is to add randomization to the layout of the address space, so an attacker has to guess the layout while trying to exploit a vulnerability. So by design, ASLR alone can't prevent memory corruption from leading to code execution, only slow it down. Finding techniques to speed up the guessing attempts of the memory layout can render ASLR practically mute.

This document will showcase three implementations of this technique:

1. Cisco Nexus Switches - Reduce the exploitation time by a factor of 15.
2. Cisco VOIP phones attempt #1 - Reduce the exploitation time by a factor of 5.
3. Cisco VOIP phones attempt #2 - One exploit that works for all ASLR options in parallel

Who we are

Armis Labs is the 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:

- **EtherOops:** Exploit Utilizing Packet-in-Packet Attacks on Ethernet Cables To Bypass Firewalls & NATs. The technical whitepaper for this research can be found here:
    - EtherOops: Bypassing Firewalls and NATs By Exploiting Packet-in-Packet Attacks in Ethernet
- **CDPwn:** 5 Zero-Day vulnerabilities in various implementations of Cisco's CDP protocol, used by a wide array of their products. The technical whitepaper for this research can be found here:
    - CDPwn: Breaking the discovery protocols of the Enterprise-of-Things
- **URGENT/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 devices

Exploit mitigations in CDPwn-vulnerable devices

We've chosen two study cases devices on which we attempted the bypass of exploit mitigations on CDPwn-vulnerable devices. The first is a Cisco Nexus switch, vulnerable to CVE-2020-3119, a stack overflow in the processing of the Power Request TLV in the CDP daemon, and the second is a Cisco 88xx VoIP phone, vulnerable to CVE-2020-3111, a stack overflow in the processing of the Port ID TLV in the VoIP's CDP daemon.

The most potent exploit mitigation that exists in both of these devices is ASLR. As we'll see shortly, the ASLR used by these devices is Linux's default 32-bit ASLR implementation that is known to have some inherent weaknesses. Knowing this, we set out to see how we can leverage these weaknesses to bypass this exploit mitigation in these devices.

But first, some background.

Background on ASLR

ASLR stands for Address Space Layout Randomization. When trying to exploit memory corruption vulnerabilities the memory layout of the targeted process is required in order to reach code execution. ASLR introduces randomness to the process memory layout that makes these types of exploitations harder. An attacker will need to find some info-leak vulnerability in order to determine the chosen memory layout of the targeted process, use an alternative method to bypass it, or reduce the randomness that ASLR has introduced.

A process that wishes to take advantage of ASLR, needs to use an operating system that supports it. It also needs to be compiled in such a way that allows the OS to later relocate the process segments to the chosen memory layout. In GCC for example, this requires the use of the Position Independent Code (PIC) flag for shared libraries or Position Independent ELF (PIE) flag for the main executables.

In Linux, for example, there is some overhead (~15%) to a process' startup time when using PIE, so certain Linux binaries will not be compiled with it. However, in both CDP daemons found on vulnerable devices, the PIE flag is in use, and thus ASLR is supported in them.

The concept of ASLR originated with the Linux PaX project in July 2001. It was designed as a patch to the mainline Linux Kernel, but a slightly different more permissive implementation was eventually merged into Linux mainline kernel. The *grsecurity* project includes PaX patches, along with other kernel patches that aim to provide greater kernel security and auditing capabilities against vulnerability exploitation.

PaX Tux - It has an ax and shield so it's probably secure

The strength of the ASLR implementation is determined by the amount of entropy that is introduced to the system using it. For example, if we have an exploit that returns to *system* (in *libc*) and the address of the *system* function has only one random bit, one try has a 50% chance of success but with just four tries, we raise the success rate to 93.75%. If the address has one byte of randomness in the address, after one try there is only 1/256 success rate and to get the success rate above 90%, an attacker will need ~600 tries!

Linux Mainline ASLR implementation

To start analyzing ASLR from somewhere, we need to think -- which part of a process is most likely to be randomized in the address space? We know that shared libraries, for example, are relocatable ELF objects, to support being loaded into different processes, with different address spaces. So we can analyze ASLR by analyzing the loading sequence of shared libraries while loading a certain PIE executable. To do that, we will examine the *ls* command:

```
bash# file /bin/ls
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=9567f9a28e66f4d7ec4baf31cfbf68d0410f0ae6, stripped
```

The *ls* executable is dynamically linked, meaning the shared objects locations are determined on execution and the interpreter is *ld-linux*. Following an *strace* log when running *ls* reveals the following:

```
execve("/bin/ls", ["/bin/ls"], 0x7ffdc18408c0 /* 29 vars */) = 0
brk(NULL)                               = 0x55998e7d6000
...
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f0272fb3000
mprotect(0x7f027319a000, 2097152, PROT_NONE) = 0
mmap(..., 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = ...
mmap(..., 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = ...
close(3)
...
```

The trace starts with *execve*. At this point the kernel loads the *ls* and *ld.so* binaries into the new process memory. Right after that, *ld.so* loads the needed library, declared in the *ls* ELF file. To do so, it uses *mmap* to allocate memory for each of the libraries segments (read/write section for the data segments, read/execute for the code segment, etc.). So, what is actually implementing ASLR in this flow?

The answer is in the kernel. Looking at an *mmap* implementation in a recent Kernel leads to the conclusion that *mmap* **allocations are sequential**, starting from *mmap_base* (*mmap_compat_base* for 32-bit processes).

That **base** for *mmap* memory allocation is randomized on process start, which occurs inside the above call to *execve* -- which also deals with implementing ASLR for the various segments of the main executable itself. The base for *mmap* is chosen using the function *arch_pick_mmap_layout*. The function initializes both process variables - *mmap_base* (used for 64-bit processes) and *mmap_compat_base* (used for 32-bit processes):

```
void arch_pick_mmap_layout(struct mm_struct *mm, struct rlimit *rlim_stack)
{
...
       arch_pick_mmap_base(&mm->mmap_base,
                   &mm->mmap_legacy_base,
                     arch_rnd(mmap64_rnd_bits), task_size_64bit(0),
                     rlim_stack);

#ifdef CONFIG_HAVE_ARCH_COMPAT_MMAP_BASES
       /*
        * The mmap syscall mapping base decision depends solely on the
        * syscall type (64-bit or compat). This applies for 64bit
        * applications and 32bit applications. The 64bit syscall uses
        * mmap_base, the compat syscall uses mmap_compat_base.
        */
       arch_pick_mmap_base(&mm->mmap_compat_base,
                   &mm->mmap_compat_legacy_base,
                     arch_rnd(mmap32_rnd_bits), task_size_32bit(),
                     rlim_stack);
#endif
}
```

Both functions call *arch_pick_mmap_base* using *arch_rnd* to determine the random bits of the *mmap_base*. The parameter to *arch_rnd* is *mmap64_rnd_bits* (20 by default) for 64-bit processes and *mmap32_rnd_bits* (8 by default) for 32-bit processes. *arch_rnd* randomizes the requested number of bits and aligns the result to the *PAGE_SHIFT* (4KB by default):

```
static unsigned long arch_rnd(unsigned int rndbits)
{
	if (!(current->flags & PF_RANDOMIZE))
		return 0;
	return (get_random_long() & ((1UL << rndbits) - 1)) << PAGE_SHIFT;
}
```

Since our target devices both use the 32-bit version of ASLR in Linux, the key takeaways we can deduce from the above logic (in the default settings, used by our target devices) is this:

- The difference between two ASLR options is the same for all loaded shared libraries since only the base is randomized
- The default number of random bits for 32-bit processes is 8 and the page alignment is of 4 KB so the max difference between two ASLR options is 1MB.
- **There is no randomization for the gap size between adjacent shared objects. The entire process memory is just shifted between ASLR options, meaning that any info-leak that reveals an address in any shared object reveals the addresses of all shared objects, including libc!**

Linux users can change the number of random bits from user mode by setting:

- */proc/sys/vm/mmap_rnd_bits* - Changes the number of random bits for 64-bit processes
- */proc/sys/vm/mmap_rnd_compat_bits* - Changes the number of random bits for 32-bit processes

ASLR bypass - Reduce exploitation time

In the case of our target device, there are only 256 options for the memory layout of the target process (the CDP daemon). By systematically trying a certain ASLR option again and again, an attacker has $P = \frac{1}{256}$ chance for successful exploitation on each attempt. The average number of tries until successful exploitation is $\frac{1}{P} = 256$.

As stated earlier, by design, ASLR is not meant to prevent code-execution all together. If an attacker has the ability to trigger a memory corruption vulnerability numerous times, he can simply brute force his way, until he stumbles on the correct ASLR shift. The question that remains is -- how much time would it take to bypass ASLR in the target devices, only by trying again and again?

Exploitation time for Nexus switches (NX-OS)

NX-OS is the operating system used by Cisco Nexus switches. Generally speaking, it is a Linux operating system with some modifications.

The CDP daemon running in the Nexus switches (*cdpd*) is re-launched every time it crashes (which would occur if the **wrong** ASLR shift was guessed). However, the Nexus switch reboots if the *cdpd* daemon crashes more than two times in a five minute period and takes approximately 3.5 minutes to boot.

Thus, the average time for a successful attack **without** crashing the device is:

$$\frac{1}{P} / Attempts\_without\_crash \times Time\_between\_attempts = 256 / 2 \times 5 = 640\ Minutes \approx 10.5\ Hours$$

There is a reverse linear correlation between *P* and the exploitation time.

Exploitation time for Cisco 88xx VoIP phones

The 88xx VoIP phones crash each time the CDP daemon crashes and take about a minute to reboot. Thus, the average time for a successful attack is ~ **256 minutes** = **4 Hours**, but during this time, the device is in a constant boot loop.

So despite the fact ASLR is **not** efficient in our target device to prevent exploitation from being practical (an attacker **can** wait a couple of hours to get to code execution if needed...), it will require a lengthy process that will probably be detected by IT or any other user that notices the continuous reboot of the targeted device.

The takeaway from this section is that to make this brute force attack more practical, we need to find a way to make *P* bigger.

P	NX-OS average exploitation time (~)	VOIP average exploitation time (~)
$\frac{1}{256}$	10:30 Hours	4 Hours
$\frac{5}{256}$	02:15 Hours	1 Hour
$\frac{10}{256}$	01:00 Hours	25 Minutes
$\frac{15}{256}$	40 Minutes	17 Minutes

NX-OS Stack Overflow

Recap - CVE-2020-3119 - Stack Overflow in the Power Request TLV

The vulnerability resides in the processing function for Power Request TLVs - a CDP TLV frame made for negotiation of Power-over-Ethernet parameters.

The Power Request TLV contains a list of requested power specifications. The list's 16-bit length field is not validated correctly and is used to copy the list to a fixed size buffer allocated on the stack and a fixed offset from an additional pointer (*a1*, which is also allocated 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] = ntohl(*current_offset);
    ...

    // Write, What, Where primitive since a1 is on the stack

    a1->levels[counter] = ntohl(*current_offset);
    ++current_offset;
    ++counter;
  } while (counter != length + 1);
}
```

Decompiled code snippet from the CDP parsing function

There are no stack cookies in use in this process (Why???), so exploiting this function gives the attacker two primitives:

1. **Code execution:** Using the stack overflow, the attacker overwrites the return address and can jump to arbitrary code.
2. **Write-What-Where:** By setting a1, the attacker can write arbitrary data to arbitrary addresses.

In the general case of stack overflow exploitation, when DEP (Data Execution Prevention) is in use, one would create an ROP (Return Oriented Programming) chain that will set-up a call to *libc*'s *system* function, using an ROP gadget that will call *system* with some command to execute (that can be set up on an overwritten stack variable). However, since the Write-What-Where primitive is triggered in this flow before code execution is achieved, the *a1* register needs to be pointed to a valid read/write memory so that the process will not crash.

Such memory space is found in the data section of shared objects. Combined with this primitive, a simple ROP chain can be used to call the *system* function with an attacker-defined command.

Putting it all together, the following ROP chain can be constructed, when ASLR is **disabled**:

Offset(from $esp)	Variable	Data
-0x40	temp	
0x00	Saved registers	
0x04	Return address	ROP gadget: pop eax ret
0x08	a1	Some read/write memory, for example, the data section of some shared object
0x0C	Call to system	Address of libc *system* function
0x10		Next ROP gadget
0x14	Argument to *system*	Address of the command to execute on the stack
0x18		Command to execute ("echo CDPwn")

ASLR characteristics for the CDP daemon in NX-OS

Examining the ASLR characteristics specifically of the CDP daemon in NX-OS reveals that in fact the amount of entropy in the ASLR implementation is quite limited. Despite the fact the OS is a 64-bit Windriver Linux, the CDP daemon is a 32-bit process, so it can't utilize the full power of 64-bit ASLR and thus the 32-bit ASLR is in use.

Calling *ldd* repeatedly for the *cdpd* daemon and examining the shared objects addresses reveals these expected key features:

- There are only 256 ASLR options (The default *mmap32_rnd_bits* = 8 is used)
- ASLR options differ by 4KB (The default *PAGE_SHIFT* = 4KB is used)
- The space between adjacent libraries is at most 4KB (As expected)

The most straightforward way to bypass ASLR will be to find some information leak vulnerability that will tell us which ASLR option was chosen. That would be ideal but since CDP is mostly one-directional, it seems unlikely to find a vulnerability that can lead to a layer-2 packet being sent back to the attacker with some leak of the memory layout.

Making the world a better P

We needed to find a way to make P (The probability one exploit attempt succeeds) higher or completely remove the randomness introduced by the ASLR. We came up with a new technique to better the odds of reaching code execution, by finding addresses that point to a valid ROP gadget for **multiple** ASLR options at a time.

PolyROP - ASLR based switch case

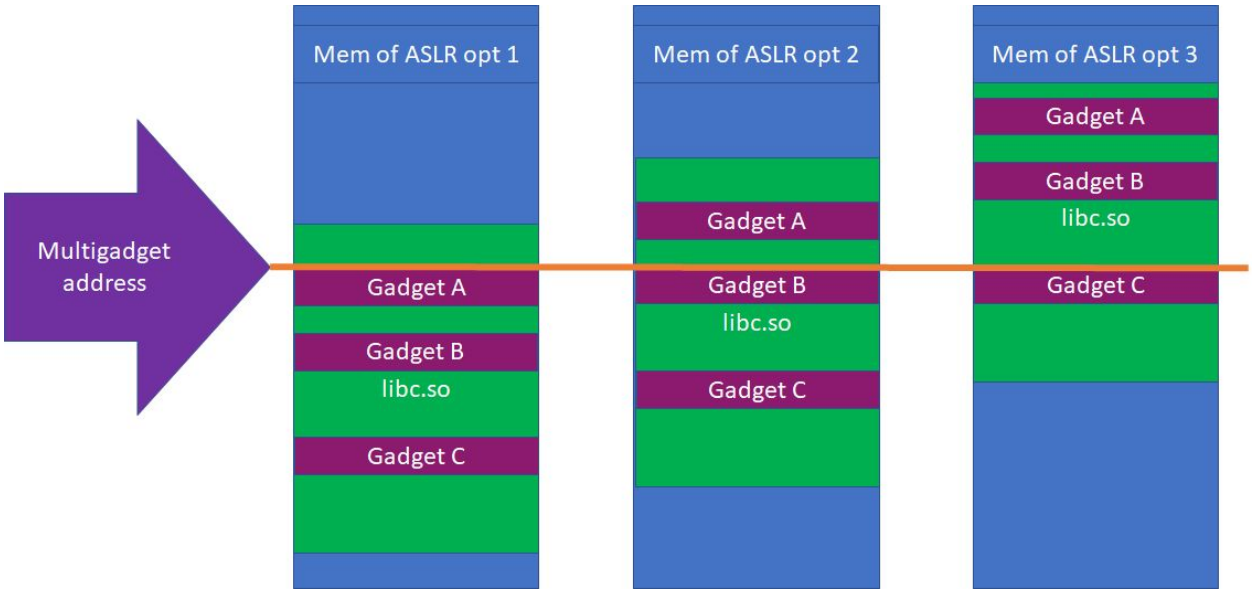
The idea is to find addresses that point to a valid ROP gadget for multiple ASLR options. For example, if we examine the *libc* binary and look at these 3 addresses:

Gadget	A	B	C
Address	0x2A314	0x67314	0x6F314
Code	2a314: pop %ebp 2a315: ret	67314: pop %ebx 67315: pop %esi 67316: ret	6f314: jmp 0x6f222 ... 6f222: add $0x10,%esp 6f225: pop %ebx 6f226: pop %esi 6f227: pop %edi 6f228: pop %ebp 6f229: ret
Stack diff	4 (One pop)	8 (Double pop)	32 (0x10 + 4 * 4)

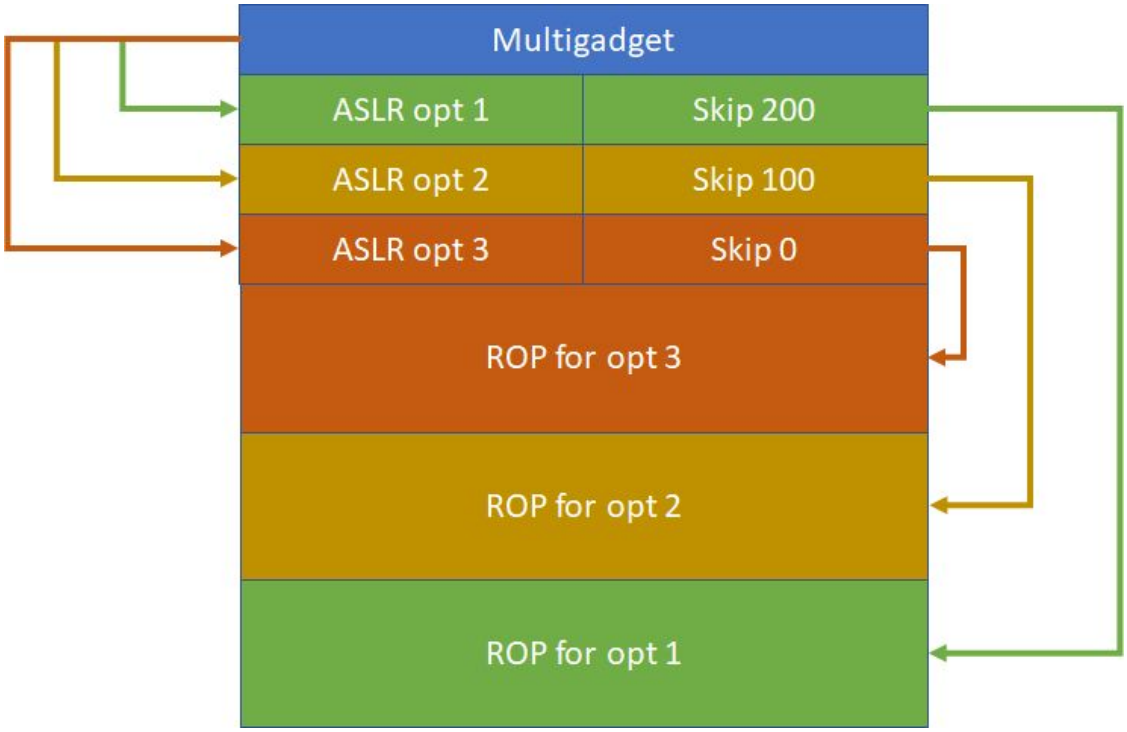
The above addresses align with three different ASLR options:

- ASLR option with offset 0x00 - 0x6F314 (Target address)
- ASLR option with offset 0x08 - 0x6F314 - (0x08 * PAGE_SHIFT) = 0x6F314 + (0x08 * 0x1000) = 0x67314
- ASLR option with offset 0x45 - 0x6F314 - (0x45 * PAGE_SHIFT) = 0x6F314 - (0x45 * 0x1000) = 0x2A314

If we set this address (0x6F314) as the overwritten return address using the stack overflow, we can make an ROP chain that will work for multiple ASLR options. The following drawing tries to illustrate this:



This first ROP gadget will act as a **multigadget**, meaning it will run a different code for each of the supported ASLR options (and will crash for any other option). The multigadget presented above was chosen in such a way that each of the individual gadgets impacts the stack in a different way. So after the multigadget is executed, each ASLR option can run a **separate** ROP chain, set up in the stack-frames that follow the chosen gadget. The ROP flow looks something like this:



Here is an example of an ROP based switch case for two ASLR options - the same address triggers a different gadget on each ASLR option:

Offset (from $esp)	Variable	Gadget Address	ASLR option 1 (0x45)	ASLR option 2 (0x08)
-0x40	temp			
0x00	Saved registers			
0x04	Return address	0x0006F314	**POP_AND_RET**	**POP_8_AND_RET**
0x08	a1	0x01080000	Read/write memory in the data section of some shared object	
0x0C		0x0100AA00	POP_20_AND_RET	
0x10		0x0103DB00		Libc *system* function address for ASLR option 2
0x14				
0x18		0x01080500		Argument for *system* function call. Read/write memory in the data section of some shared object plus needed offset to the command start.
...		...		
0x2C		0x01000B00	Libc *system* function address for ASLR option 1	
0x30				

0x34		0x01080500	Argument for *system* function call. Read/write memory in the data section of some shared object plus needed offset to the command start.	
0x38			Command to execute ("echo CDPwn; rm -rf /")	
...				

With this simple multigadget, this exploit will work with a probability of $\frac{2}{256}$ - which will result in **a 50% decrease in exploitation time**. We wrote a script to automatically find multi-gadget addresses and arrange them together to a PolyROP exploit.

Note: The exploit above leverages the write-what-where primitive for simplification purposes. It does not depend on it to reach controlled code execution

Finding MultiROP gadgets

To test the limits of this PolyROP technique, we needed to find as many ROP gadgets as we could. These would later be used to mix and match with different ASLR options. To do so, we mapped all *ret* instructions and went up until we hit a memory changing opcode:

```
6f2d2: mov    %esi,(%eax) # Invalid gadget address since memory is used
6f2d4: add    $0x10,%esp  # Valid gadget address
6f2d7: mov    %ecx,%eax   # Valid gadget address
6f2d9: pop    %ebx        # Valid gadget address
6f2da: pop    %esi        # Valid gadget address
6f2db: pop    %edi        # Valid gadget address
6f2dc: pop    %ebp        # Valid gadget address
6f2dd: ret                # Valid gadget address
```

Since our shellcode will end with a call to *system*, we don't have to restore execution. This means we can tolerate opcodes that alter register values. For each ROP gadget, we needed to count any changes to the stack for later use in the multigadget switch case. For example, in the above gadget, the stack is incremented by 0 for address 0x6f2dd, 4 for address 0x6f2dc, 8 for address 0x6f2db, and so on. In a similar approach, we also registered all *jmp* instructions that point to a valid ROP gadget.

Ultimately, using the above technique on *libc* library used by the CDP daemon, led us to find ~800,000 addresses that point to valid ROP gadgets for multiple ASLR options. 47 of those were simultaneously valid for **15** different ASLR options.

NX-OS Stack Overflow ASLR bypass results

Using the above-mentioned technique, we were able to develop **an exploit that works for 15 ASLR options, reducing the average exploit attempts number to ~17 (from 256)** making the exploitation time **~20 Minutes** if we are allowed to crash the device and **~40 Minutes** if we can't crash it (two attempts every five minutes).

Cisco VoIP Phones Stack Overflow

Recap - CVE-2020-3111 - Cisco VOIP Phones Stack Overflow in PortID TLV

The stack overflow vulnerability is in the parsing of PortID TLV (0x03). There are no boundary checks on the length of this TLV and the value is copied to a fixed size 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

Just as with the previous vulnerability, a simple ROP chain can be made to exploit this vulnerability and call the *system* function with the argument taken directly from the stack. But, similar to NX-OS, the Cisco VOIP phones also use ASLR.

ASLR characteristics for the CDP daemon on Cisco VOIP phones

In Cisco's 88xx VoIP phones the OS is Linux and the CPU is 32-bit Arm. Despite the different CPU architecture, the ASLR characteristics for the CDP daemon are similar to those of the NX-OS switch, since both use Linux's default ASLR implementation in 32-bit processes.

Executing the *cdp* daemon repeatedly and examining the shared objects addresses reveals these key features:

- There are only 256 ASLR options (The default *mmap32_rnd_bits* = 8 is used)
- ASLR options differ by 4KB (The default *PAGE_SHIFT* = 4KB is used)
- The space between adjacent libraries is at most 4KB (As expected)
- Threads are used and since the thread stack is allocated using mmap, **it's in a constant offset from the shared libraries.** The threads stack size is 8 MB which makes a great location for write operations since the max ASLR shift is 1 MB.

As with the Nexus switch, since CDP is mostly one-directional, searching for info-leaks seemed like an impractical option.

One exploit to rule them all (ASLR options)

Let's examine the epilogue of the overflowed function:

```
SUB     SP, R11, #0x28 ;
LDMFD   SP, {R4-R11,SP,PC} ;
```

In this function epilogue, the SP itself is restored from the stack, before the function returns to its caller (by popping the return address from the stack to PC). So to control the PC, we also need to overwrite the stack pointer. In the Cisco Nexus case, our PolyROP technique relied on a ROP **chain** that started from a multigadget, but then progressed to multiple chains per ASLR option. Like all segments of the target process, the stack itself is also located in a randomized address, so we can't know what value the SP should have to be able to use the stack for **any** type of ROP chain. Meaning the only shot we have to gain code execution is through a single ROP gadget.

Arm32 PolyROP - Take 1 - Register branching

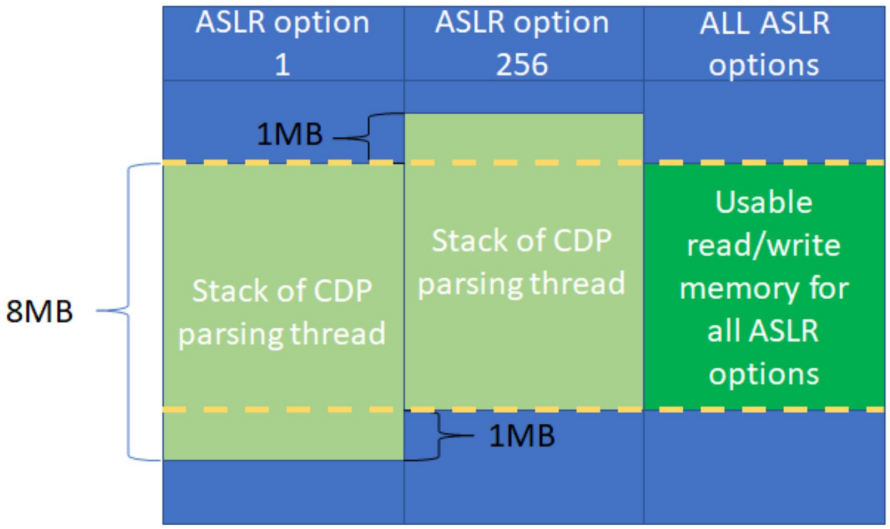
To overcome the above limitation, an attacker would need to find a gadget that jumps to a different attacker-controlled location, based on the chosen ASLR option, without relying on any stack data to continue the ROP chain (since we unwillingly overwrite SP itself). In the above function epilogue we notice that 8 registers (other than SP and PC) are overwritten by our overflow. Using these registers as a way to continue the code flow after our single ROP gadget, was an idea worth exploring.

The CPU architecture of the device is ARM-32 and the *BLX* opcode can be used to achieve this goal. BLX R* jumps to the address pointed by the selected register—so, by finding an address that branches to different registers on different ASLR options, an attacker could have one call to a fully controlled function address.

An example for such an address is *0x**9f0* of *libc*:

Address	0x329F0	0xDB9F0	0xFF9F0
Code	BLX R10	BLX R8	BLX R6

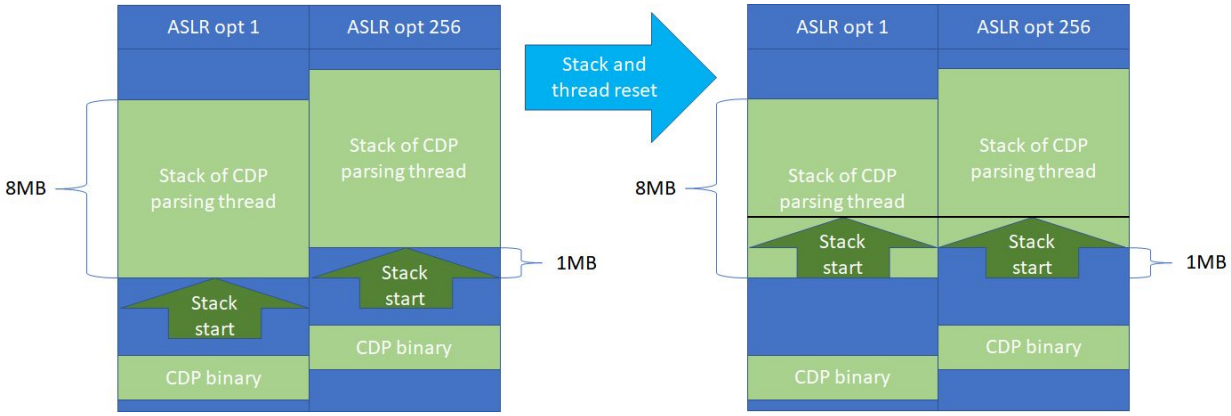
Gaining controlled code execution with a single function call might seem a bit far-fetched. The idea was to use this single function call to restart the CDP parsing thread by calling the first function in this thread (*dpConficCmdThrd*). When the thread is restarted, a second attempt to exploit the vulnerability can be taken, only this time location of the the stack would be predictable, since the SP was overwritten in the first attempt. Selecting a valid stack address for the different ASLR options is also doable, since the original stack is 8MB long, while the maximum ASLR shift is only 1MB.



An exploit scenario would look like this:

1. Trigger the stack overflow that will move the stack to a new location and restart the CDP parsing thread.
2. Trigger the stack overflow a second time, this time the exact location of the stack is known and a full ROP chain can be used..

When sending the second packet, the attacker knows the exact location of the stack, so a simple PolyROP can be used (same as the one made for the NX-OS).



This technique is limited by the number of registers under the attacker's control (R4-R11) so the theoretical limit here is 7 options ➔ Not good enough.

Arm32 PolyROP - Take 2 - Stacking ASLR options

Further examining the overflown stack, another primitive was discovered - A **relative write** of 0x154 bytes, of which 0xF3 bytes are attacker-controlled. This can be achieved by overflowing stack variables that contain pointers that are later used to copy some attacker-controlled data from the input packet. The relevant stack variables are located before the end of the stack-frame, and so the stack pointer, and PC can be left untouched, and the CDP process will not crash. The vulnerable *memcpy* call:

```
memcpy(&cdpCache + 0x7F8 * port + 0x154 * entry, main_buffer, 0x154);
```

- *port* and *entry* are attacker-controlled using the stack overflow.
- *cdpCache* is a global variable in the *.bss* section of the *cdp* binary.
- *main_buffer* is attacker-controlled from offset 0x61.

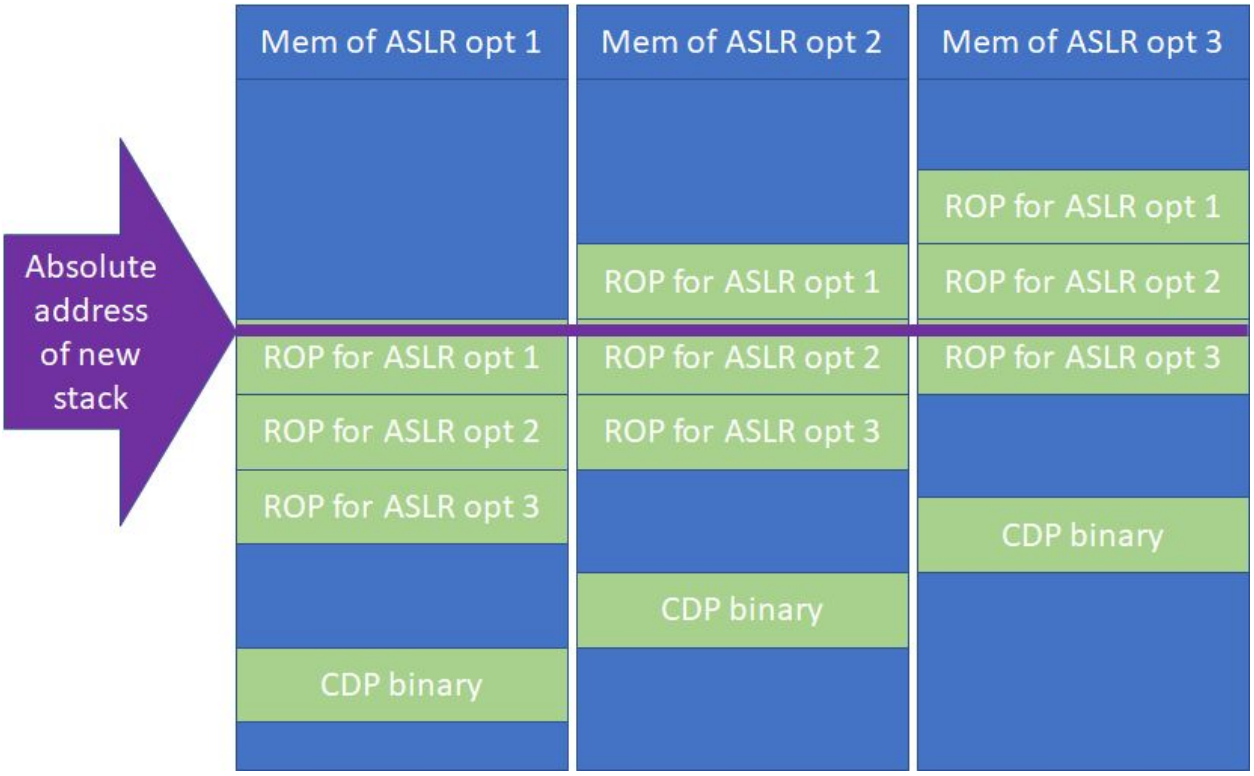
By setting *port* and *entry* correctly, one could write almost anywhere in the process memory. Also, since this is a relative write primitive, it can be used in a deterministic fashion, ignoring ASLR all together. It's important to note that there is also an integer overflow in the above line, making any offset writable using this primitive.

With the help of this primitive, we could prepare a stack of ROP chains for every ASLR option in the process memory and eventually pivot the stack to an absolute address. **Since the write is relative and the pivot is absolute, the combination completely negates the ASLR randomness, making the exploitation success rate 100%.**

Consider the following scenario:

1. The attacker writes the stack of ROP chains for ASLR option 1 in offset -0x4000, the stack ROP for ASLR option 2 in offset -0x3000 and stack ROP for ASLR option 3 in offset -0x2000. All of these writes are achieved using the relative write primitive without overwriting the stack pointer or the *PC*. Each write is achieved using one specially crafted CDP packet.

2. The attacker sends another specially crafted CDP packet, this time, overwriting the stack pointer with an **absolute address**. That absolute address holds the correct stack for the current ASLR option because the stacks were written using a **relative write**.



Once the stack is pivoted, all addresses are known since the ROP chain on the pivoted stack is built for the specific ASLR option that triggered it.

Cisco VOIP phones Stack Overflow ASLR bypass results

Using the above-mentioned technique, we were able to develop **an exploit that works for all ASLR options without crashing the device even once.** Moreover, this vulnerability can be exploited using an Ethernet broadcast packet. This means an attacker could send 257 broadcast packets and **take control over all vulnerable VoIP phones in the LAN in parallel.**

Conclusion

This exploit development experiment underlines the fragility of exploit mitigations. On paper, they may seem quite effective. Sending a blind CDP packet that will trigger a memory corruption, to a device with a **random** address space layout can sound unexploitable. However, leveraging the limitations in the randomness of this mitigation, as found in the vulnerable Cisco devices, combined with strong primitives that arise from the vulnerability itself (relative write, for example), can render the ASLR mitigation useless.

From a more high-level perspective, understanding that the CDPwn vulnerability may be turned into an extremely powerful exploit, that has the ability to remotely, without any authentication or user interaction, **take over all vulnerable VoIP phones** in an organization's network, in a broadcast attack that works by simply sending 257 packets to the network is quite astonishing. This type of ability is remarkable since the attacker doesn't even have to do any reconnaissance steps to find target devices within the network. Once the VoIP phones have been attacked, the executed shellcode can simply connect back to an attacker's C&C server, and await further instructions.

VoIP phones have already been targeted in attacks in the wild, either in order to preserve a hold in targeted networks or in order to eavesdrop on corporate calls. The importance to secure these types of devices, alongside the switches and routers that were also found vulnerable to CDPwn, becomes even more apparent, considering the effectiveness of a potential exploit that could leverage these vulnerabilities, as demonstrated in this research.

ABOUT ARMIS

Armis is the first agentless, enterprise-class security platform to address the new threat landscape of unmanaged and IoT devices. Fortune 1000 companies trust our unique out-of-band sensing technology to discover and analyze all managed, unmanaged, and IoT devices—from traditional devices like laptops and smartphones to new unmanaged smart devices like smart TVs, webcams, printers, HVAC systems, industrial robots, medical devices, and more. Armis discovers devices on and off the network, continuously analyzes endpoint behavior to identify risks and attacks, and protects critical information and systems by identifying suspicious or malicious devices and quarantining them. Armis is a privately held company and headquartered in Palo Alto, California.

20201214-1