VIEW BLOG POST

ZTE Router Vulnerabilities

How an attacker could exploit two vulnerabilities to gain full control

PREPARED BY MARCIN NOGA

Updated February 22, 2022

TALOS
Cisco Security Research

CONTENTS

INTRODUCTION

ZTE recently patched seven vulnerabilities Cisco Talos' vulnerability research team discovered the ZTE MF971R wireless hotspot and router. Two of those vulnerabilities, CVE-2021-21748 and CVE-2021-21745, could be exploited to access the device's root shell and execute arbitrary code. In this paper, we'll walk through our process for discovering these vulnerabilities and how an attacker could exploit them in a worst-case scenario.

The ZTE MF971R is a portable router with Wi-Fi support and that also works as an LTE/GSM modem. Users can manage the router's settings via a web panel that requires authentication. An attacker could exploit the aforementioned vulnerabilities to build a stable exploit and gain remote root access on the ZTE MF971R device just after a user — authenticated or not — visits a malicious site.

RECON

First, we should look at what the device and its settings panel looks like (Figures 1 and 2).

This is not the first time security researchers have found a way to access wireless hotspots: This DEFCON talk showed a way an attacker could exploit a ZTE MF910, an older model of the MF971R, to gain root access.

ANDROID DEBUG BRIDGE (ADB)

If I could turn on ADB on this device, I could access the Unix shell and potentially transfer files from and to the device, among other actions.

As "G Richter" found in his research into the MF910, there are a few ways to achieve this, but none of them worked for me (Figure 3).

As it later turned out, all of the "_set_" commands should be called with an additional unique AD parameter sequence number. But even with that parameter, I could not activate ADB.

HARDWARE DETAILS

After a dozen failed attempts to turn on ADB, I decided to look for more conventional ways to access the firmware. I started by looking at the internal hardware of the device (Figure 4).

Figure 1. *ZTE MF971R.*

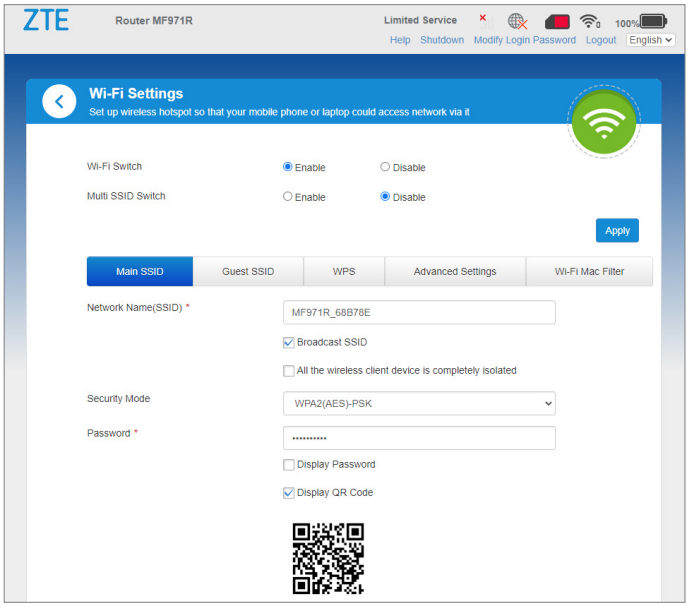

Figure 2. *MF971R web control panel.*


```
Command Prompt

C:\Users\icewall>curl -i --referer http://127.0.0.1 "http://192.168.2.1/goform/goform_get_cmd_process?cmd=loginfo"
HTTP/1.1 200 OK
Server: WebServer-Webs
Pragma: no-cache
Cache-Control: no-store
Content-Type: text/html
X-Frame-Options: sameorigin
X-XSS-Protection: 1; mode=block

{"loginfo":"ok"}
C:\Users\icewall>curl -i --referer http://127.0.0.1 "http://192.168.2.1/goform/goform_set_cmd_process?goformId=SET_DEVICE_MODE&debug_enable=2"
HTTP/1.1 200 OK
Server: WebServer-Webs
Pragma: no-cache
Cache-Control: no-store
Content-Type: text/html
X-Frame-Options: sameorigin
X-XSS-Protection: 1; mode=block

{"result":"failure"}
C:\Users\icewall>curl -i --referer http://127.0.0.1 "http://192.168.2.1/goform/goform_set_cmd_process?goformId=USB_MODE_SWITCH&usb_mode=6"
HTTP/1.1 200 OK
Server: WebServer-Webs
Pragma: no-cache
Cache-Control: no-store
Content-Type: text/html
X-Frame-Options: sameorigin
X-XSS-Protection: 1; mode=block

{"result":"failure"}
C:\Users\icewall>curl -i --referer http://127.0.0.1 "http://192.168.2.1/goform/goform_set_cmd_process?goformId=MODE_SWITCH&switchCmd=FACTORY"
HTTP/1.1 200 OK
Server: WebServer-Webs
Pragma: no-cache
Cache-Control: no-store
Content-Type: text/html
X-Frame-Options: sameorigin
X-XSS-Protection: 1; mode=block

{"result":"failure"}
C:\Users\icewall>
```

Figure 3. *Failed attempt to activate ADB server.*

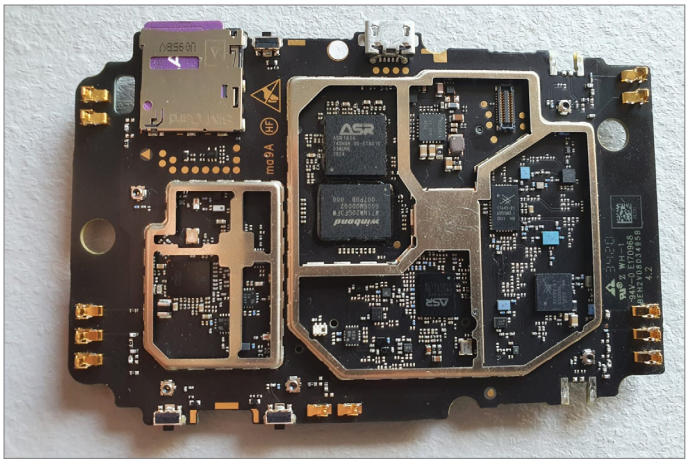

Figure 4. *MF971R board.*

I noticed the following individual components:

- Qualcomm® MDM9230
- ASR 1826/Marvell Unveils ARMADA Mobile PXA1826 5-Mode 4G LTE Release 10 Modem
- Winbond W71NW20GF3FW
- ASR RF865

UART

After getting more familiar with the board, Claudio Bozzato (who helped me with the hardware part of this project) noticed some interesting pads (Figure 5).

Further examination found that those pads are part of the UART interface (Figure 6).

Knowing that, I could use BusPirate to reach a shell console on the device (Figure 7).

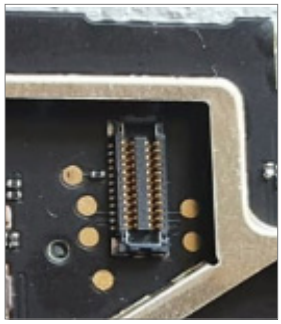

Figure 5. *Part of MF971R board presenting UART pads.*

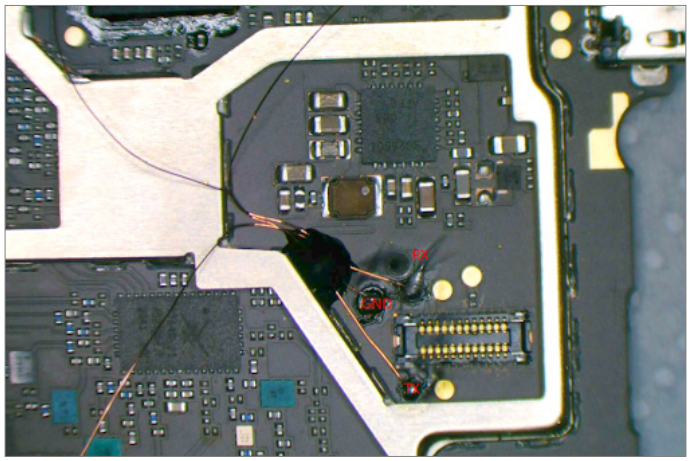

Figure 6. *UART pads.*

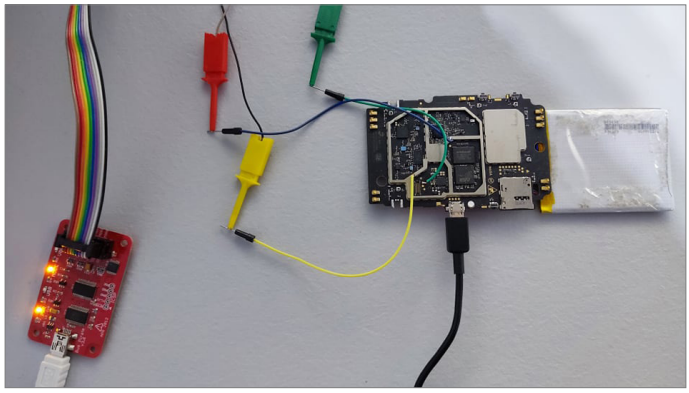

Figure 7. *BusPirate connected to ZTE 971R UART port.*

Using Putty as a terminal, we can pull up the screen shown in Figure 8.

There's no authorization needed, so I had direct access to the root shell and can start my research.

THE GOAL

The goal was simple: Discover as many bugs as possible. Beside that, I wanted to try to see whether we could find a chain of bugs that would allow a hypothetical attacker to gain the ability to execute remote code on the device without having any existing pre-conditions, such as knowing the credentials of the web panel or a targeted user already being logged in.

BUG HUNTING

The goal was simple: Discover as many bugs as possible. Beside that, I wanted to try to see whether we could find a chain of bugs that would allow a hypothetical attacker to gain the ability to execute remote code on the device without having any existing pre-conditions, such as knowing the credentials of the web panel or a targeted user already being logged in.

FIRMWARE DUMP

Having a root shell over UART and internet connectivity on the device, we could transfer files in a few different ways, but I wanted to activate the ADB server and copy the necessary files using that communication channel.

Figure 8. *Root shell on MF971R via UART connection.*

After playing a bit with the device (Figure 9) and using my knowledge from previous research, I decided to go after the HTTPD daemon and its related functionalities (Figure 10), which seem to have the biggest attack surface.

To do this, I copied `zte_topsw_goahead` binary where HTTPD is implemented, along with all important shared libraries, and started the process of reverse-engineering.

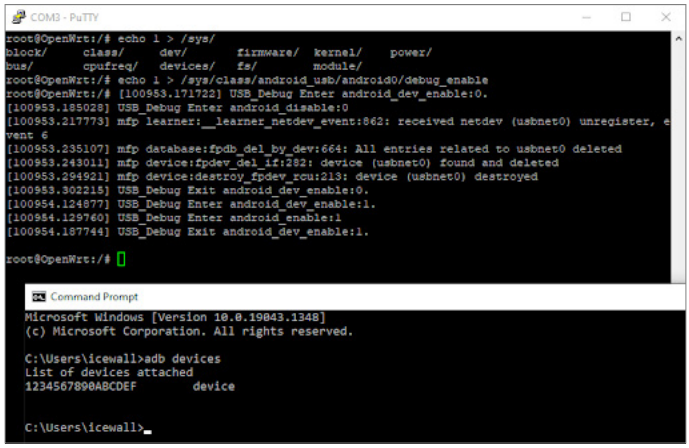

Figure 9. *Setting the "1" flag in "debug_enable" activates the ADB server on the device and makes it visible to the ADB client on Windows PC via USB connection.*

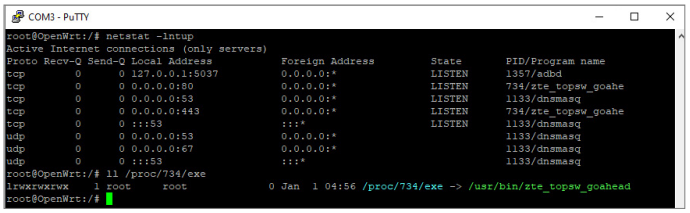

Figure 10. *Netstat output showing all available network daemon on default configuration.*

REVERSING ZTE_TOPSW_GOAHEAD

For people familiar with the embedded world, even the name of this file suggests the technology/project used to create HTTPD daemon: a GoAhead server. Because it's an open-source solution, we can easily get into its internals, learn about its architecture and important APIs.

There are few branches of this project, so it's good we figured out at the beginning which version we are dealing with. Looking around in a current branch 5.x, I found a function called "websSetEnv" where a constant value represents the current server version (Figure 11).

Knowing that fact, we searched for one of the constant strings used in this function, e.g, "GATEWAY_INTERFACE" in zte_topsw_goahead binary (Figure 12).

Figure 12 shows that the current version is 2.5.0, which allows us to browse the code in a proper GitHub branch and change the zte_topsw_goahead binary function names, definitions and add important structures like "webs_t," which will make our bug hunting process easier.

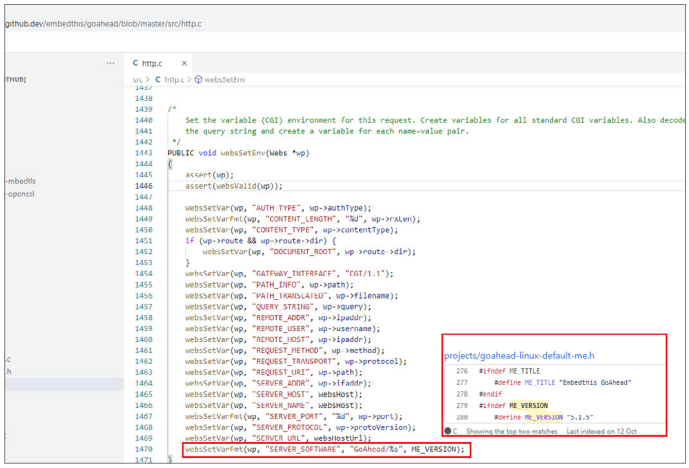

Figure 11. *The definition of the function websSetEnv in GoAhead 5.x.*

```
1
2  void FUN_0001922c(int param_1)
3
4  {
5    char *pcVar1;
6    char *pcVar2;
7    size_t sVar3;
8    int iVar4;
9    char *local_28;
10   undefined4 local_24;
11   undefined auStack32 [8];
12
13   FUN_00018e5c(param_1,"QUERY_STRING",*(undefined4 *)(param_1 + 0xc4));
14   FUN_00018e5c(param_1,"GATEWAY_INTERFACE","CGI/1.1");
15   FUN_00018e5c(param_1,"SERVER_HOST",&DAT_0012618c);
16   FUN_00018e5c(param_1,"SERVER_NAME",&DAT_0012618c);
17   FUN_00018e5c(param_1,"SERVER_URL",DAT_00123718);
18   FUN_00018e5c(param_1,"REMOTE_HOST",param_1 + 0x30);
19   FUN_00018e5c(param_1,"REMOTE_ADDR",param_1 + 0x30);
20   FUN_00018e5c(param_1,"PATH_INFO",*(undefined4 *)(param_1 + 0xb4));
21   FUN_00016048(DAT_00126184,auStack32,8);
22   FUN_00018e5c(param_1,"SERVER_PORT",auStack32);
23   FUN_00018e5c(param_1,"SERVER_ADDR",param_1 + 0x50);
24   FUN_0001397c(&local_28,0xfe,"%s/%s %s/%s","WebServer-Webs","2.5.0","PeerSec-MatrixSSL",
25                "3.9.3-OPEN");
```

Figure 12. *Ghidra decompilation listing of websSetEnv function.*

Endpoints registration

We used the web panel and observed generated HTTP traffic coming through the device. Eventually, we noticed that, as in the MF910 version, there are two major

communication endpoints (Figure 13).

We specifically focused on:

/goform/goform_get_cmd_process

- Used via GET
- Parameter name used to pass values from a user:
  - cmd
  - multi_data
- Purpose: Reads current state of config/device

/goform/goform_set_cmd_process

- Used via POST/GET
- Parameter name used to pass values from the user:
  - goformId
  - depends on formId additional params
- Purpose: Changing the state of the device

Now, we'll look inside the main function where registration of these endpoints should take place. We'll also determine if there are hidden endpoints, as we saw in the MF910.

Figure 14 shows there are fewer hidden endpoints than in MF910, but we can still find registration of handlers for mentioned major endpoints (Figure 15).

What about ADB activation over HTTP?

Before we move further, let's try to figure out what exactly changed in this version so the "old tricks" to activate ADB didn't work.

Looking for the **"MODE_SWITCH"** handler, we can see the code shown in Figure 16.

This is completely missing **"system"** call related to ADB activation. Instead, we see information about a necessary key.

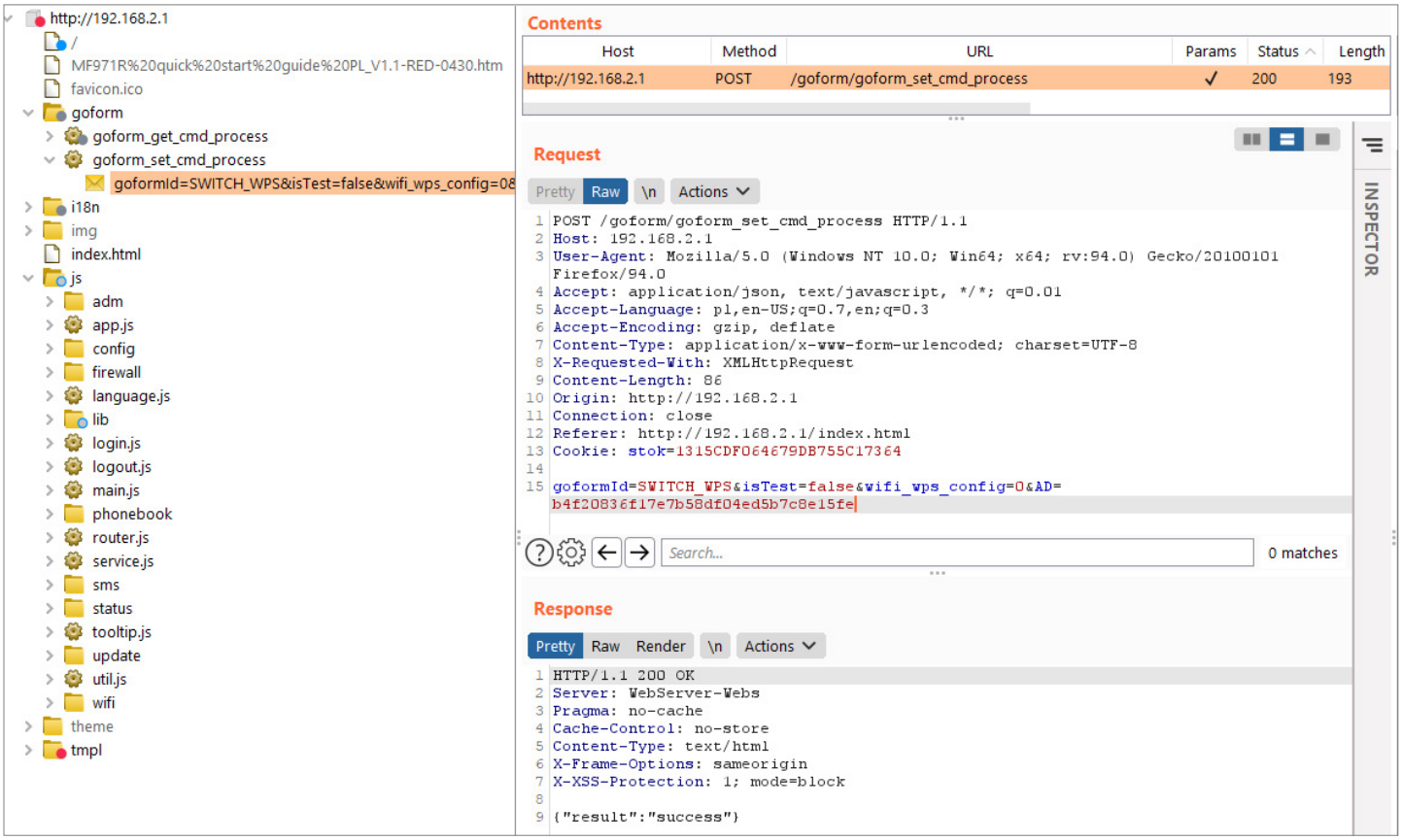

Figure 13. *POST request sent to goform_set_cmd_process endpoint.*

```
64  sub_19C3C(src);
65  sub_E928("index.html");
66  sub_141B4(&g_default_value);
67  sub_18D0C(80, 5);
68  websUrlHandlerDefine(&g_default_value, 0, 0, (int)sub_1417C, 1);
69  websUrlHandlerDefine("/goform", 0, 0, (int)goform_handler, 0);
70  websUrlHandlerDefine("/cgi-bin", 0, 0, (int)cgi_bin_handler, 0);
71  websUrlHandlerDefine("/mmc2", 0, 0, (int)mmc2_handler, 0);
72  websUrlHandlerDefine("/api/xmlclient/post", 0, 0, (int)xml_client_handler, 0);
73  websUrlHandlerDefine("/client/backup", 0, 0, (int)backup_handler, 0);
74  websUrlHandlerDefine("/api/nvramul.cgi", 0, 0, (int)nvramul_handler, 0);
75  websUrlHandlerDefine(&g_default_value, 0, 0, (int)qr_codes, 2);
76  goform_handlers_registration();             // main_goform_handlers
77  v12 = websUrlHandlerDefine("/", 0, 0, (int)root_handler, 0);
```

Figure 14. *Endpoint registration.*

```
1  int goform_handlers_registration()
2  {
3    __off_t v0; // r6
4    size_t v1; // r7
5    char *v2; // r0
6    char *v3; // r4
7    int v4; // r0
8    int v5; // r7
9    ssize_t v6; // r6
10   ssize_t i; // r2
11   ssize_t v8; // r1
12   ssize_t v9; // r5
13   const char *v10; // r1
14   signed int v11; // r5
15   int v12; // r0
16   int v13; // r0
17   char s[36]; // [sp+14h] [bp-94h] BYREF
18   struct stat v16; // [sp+38h] [bp-70h] BYREF
19
20   register_handlers("goform_get_cmd_process", (int)handler_goform_get_cmd_process);
21   register_handlers("goform_set_cmd_process", (int)handler_goform_set_cmd_process);
22   register_handlers("goform_process", (int)handler_goform_process);
```

Figure 15. *Major endpoints' registration.*

```
1  int __fastcall handler_goform_process(int web, int a2, int a3)
2  {
3    websRec *v3; // r4
4    const char *formID; // r5
5
6    v3 = (websRec *)web;
7    if ( web )
8    {
9      formID = (const char *)get_value_of_param((websRec *)web, "goformId", 0x5C86B);
10     if ( !strcmp(formID, "MODE_SWITCH") )
11     {
12       zte_syslog_append(
13         0,
14         386609,
15         863,
16         0,
17         "[ERROR] not supprot  without key. please see API zte_device_adb_mode_switch ",
18         a2,
19         a3);
20     }
21     else if ( !strcmp(formID, "ONLINE_UPGRADE_TOOL") )
22     {
23       handler_ONLINE_UPGRADE_TOOL(v3);
24     }
25     else
26     {
27       zte_syslog_append(0, 386609, 878, 0, "[ERROR]zte_goform_process -> unknown goform id:[%s].", formID);
28       responseError(v3, "failure");
29     }
30     web = wbsReturnStatus(v3, 200);
31   }
32   return web;
33 }
```

Figure 16. *The function body responsible among the others for handling MODE_SWITCH option.*

Next, we'll look for the **"SET_DEVICE_MODE"** handler (Figure 17).

The software explicitly checks if you pass value "1" for parameter "debug_enable" and tell syslog that this option is not supported.

Value "2" does not activate ADB. Users cannot activate ADB via "ADB_MODE_SWITCH," either.

```
1  void *__fastcall handler_SET_DEVICE_MODE(websRec *wp)
2  {
3    void *result; // r0
4    const char *debug_state; // r4
5    char *v4; // r1
6    websRec *v5; // r0
7    char cmd[272]; // [sp+8h] [bp-110h] BYREF
8
9    result = memset(cmd, 0, 0x100u);
10   if ( wp )
11   {
12     debug_state = (const char *)get_value_of_param(wp, "debug_enable", 378987);
13     zte_syslog_append(6, 398377, 2022, 0, "web para:[debug_enable] is [%s].", debug_state);
14     if ( *debug_state && (!strcmp(debug_state, "0") || !strcmp(debug_state, "1") || !strcmp(debug_state, "2")) )
15     {
16       if ( !strcmp("1", debug_state) )
17       {
18         zte_syslog_append(6, 398377, 2033, 0, "not supprot 1 without key. please see API zte_device_adb_mode_switch.");
19       }
20       else
21       {
22         memset(cmd, 0, 0x100u);
23         sprintf(cmd, "echo %s  > /sys/class/android_usb/android0/debug_enable", debug_state);
24         if ( !j_system(cmd) )
25         {
26           zte_syslog_append(6, 398377, 2051, 0, "zte_device_set_mode:  set mode SUCCESS.");
27           v4 = "success";
28           v5 = wp;
29           goto LABEL_12;
30         }
31         zte_syslog_append(6, 398377, 2044, 0, "zte_device_set_mode:  system command ERROR.");
32       }
33     }
```

Figure 17. *SET_DEVICE_MODE function handler.*

Mitigations

In G Richter's researcher, he used a chain of bugs to gain the ability to execute remote code:

- Found pre-auth XSS to be able to bypass referer check.
- Leaked the admin password.
- Used the leaked admin password to login and gain access to APIs.
- Used cmd injection in one of post-auth API.

Before we start checking whether that scenario is possible in the MF971R, we'll try to understand what kind of APIs we have access to based on the requirements we need to pass (Figure 18).

By analyzing the "goform_get_cmd_process" handler, we noticed that it will perform another action if:

- User is logged in – **loggedIn_and_ipCheck** function setting **loggedin_flag**
  - line 6
- Passed by the **cmd** parameter which exists on predefined list **g_commands**
  - lines 15-26
- Referer is "correct" or cmd exists on some allowed/ common list, such as function **referer_check_or_ common_cmd**
  - line 29

We can easily conclude that the widest access to APIs appear when we are logged in and calling an API from the same origin. Missing some of those constraints will limit our access.


```
1  int __fastcall handler_goform_get_cmd_process(websRec *web, int a2, int a3)
2  {
3
4    memset(s, 0, 0x40u);
5    memset(loggedin_flag, 0, sizeof(loggedin_flag));
6    loggedIn_and_ipCheck(web, loggedin_flag);
7    cmd = (char *)get_value_of_param(web, "cmd", (int)&g_default_value);
8    multi_data = (_BYTE *)get_value_of_param(web, "multi_data", (int)&g_default_value);
9
10   if ( *cmd )
11   {
12     if ( filter_param(cmd) )
13     {
14       offset = 0;
15       while ( 1 )
16       {
17         bCmdEquals = strcmp(&g_commands[offset], cmd);
18         offset += 64;
19         if ( !bCmdEquals )
20           break;
21         if ( offset == 5504 )
22         {
23           cmd_exists = 0;
24           goto LABEL_14;
25         }
26       }
27       cmd_exists = 1;
28 LABEL_14:
29       if ( (!strcmp("ok", loggedin_flag) || cmd_exists == 1) && !referer_check_or_common_cmd(web, cmd) )
30       {
31         index = (unsigned __int8)*multi_data;
32         if ( *multi_data )
33         {
34           multi_data_handler(web, (unsigned __int8 *)cmd);
35         }
36         else
37         {
38           do
39           {
40             if ( !strcmp(&g_another_commands[68 * index], cmd) )
41             {
42               (*(void (__fastcall **)(websRec *))&g_another_commands[68 * index + 64])(web);
43               return wbsReturnStatus(web, 200);
44             }
45             ++index;
46           }
47           while ( index != 49 );
48           if ( special_cmd((int)web, a2, a3, cmd) == -1 )
49             read_value_from_config(web, cmd);
50         }
51       }
52       else if ( *multi_data )
53       {
54         multi_data_handler_unauth(web, cmd);
55       }
56       else
57       {
58         single_cmd_handler_unath(web, cmd);
59       }
60     }
```

Figure 18. *Code of major _get_ handler.*

But what does it mean being logged in? How exactly is the referer checked?

Am I logged in?

Next, we'll check how the function responsible for checking if the user is logged in "loggedIn_and_ipCheck" (Figure 19).

As we can see in line 13, the "loginfo" value is read from the config and if it equals "ok",

it means that there is an active session of authenticated user. Next, in lines 20-21, we see a check related to our IP address and one that's saved during authentication.

The condition is, of course, true if they are equal. It's another layer of protection for requests sent from a different IP in an internal network. Moving forward, there is a check of session cookie at line 25. That function is a bit buggy but we won't focus on it because it's still ok when combined with previous IP checks.

An interesting scenario takes place in lines 29-36 if the user sends a request to the web server from IP address that is 127.0.0.1 or 192.168.0.1 (192.168.0.1 is a default IP

```
1  char *__fastcall loggedIn_and_ipCheck(websRec *wp, char *status)
2  {
3    int v4; // r2
4    char *result; // r0
5    const char *remote_ipAddress; // r5
6    char loginfo[20]; // [sp+10h] [bp-78h] BYREF
7    char saved_ip_address[20]; // [sp+24h] [bp-64h] BYREF
8    char entry[80]; // [sp+38h] [bp-50h] BYREF
9
10   memset(loginfo, 0, sizeof(loginfo));
11   memset(saved_ip_address, 0, sizeof(saved_ip_address));
12
13   if ( zte_nvconfig_read("loginfo", loginfo, 20) != 1 )
14     return (char *)zte_syslog_append( "zte_get_loginfo: read the nv [loginfo] fail.")
15
16   remote_ipAddress = (const char *)websGetIpaddr(wp);
17   if ( !remote_ipAddress )
18     return (char *)zte_syslog_append("zte_get_loginfo: ip_address is null.");
19
20   zte_nvconfig_read("user_ip_addr", saved_ip_address, 20);// ip address saved during authentication phase
21   if ( !strcmp(loginfo, "ok") && !strcmp(remote_ipAddress, saved_ip_address) )
22   {
23     if ( !g_web_is_support_cookie )
24       return strcpy(status, "ok");
25     result = (char *)session_cookie_check(wp);
26     if ( result )
27       return strcpy(status, "ok");
28   }
29   else
30   {
31     if ( !strcmp(remote_ipAddress, "127.0.0.1") )
32       return strcpy(status, "ok");
33     result = (char *)strcmp(remote_ipAddress, "192.168.0.1");
34     if ( !result )
35       return strcpy(status, "ok");
36   }
37   return result;
38 }
```

Figure 19. *loggedIn_and_ipCheck function body.*

address of this router – you might observe different ones from the screenshots we use in this paper, but I changed it intentionally for research purposes). Then, the user is automatically considered an authenticated user and does not need to pass any additional constraints.

The implication here is simple: We can build our exploit chain like G Richter did in the case of the MF910 by finding a pre-auth XSS, abuse it to leak the admin password and then authenticate to increase the attack vector to post-auth available APIs. Before we start doing anything in that direction, we'll look for the next mitigation.

CSRF protection/Refferer check

This mitigation is implemented in a function I called "referer_check_or_common_cmd" (Figure 20).

There are more checks than only the one related to the referrer. The while loop in lines 9 - 16 runs checks to see if the passed value in **`cmd`** parameter is one of the basic values visible on the list. If so, the referrer does not need to

```
1  int __fastcall referer_check_or_common_cmd(websRec *web, const char *cmd)
2  {
3    int result; // r0
4    int v5; // r4
5
6    if ( referer_check(web) || (websGetRequestFlags(web) & WEBS_AUTH_DIGEST) != 0 )
7      return 0;
8    v5 = 0;
9    while ( 1 )
10   {
11     result = strcmp(g_basic_info[v5++], cmd);   // imei, wa_inner_version, integrate_version
12     if ( !result )
13       break;
14     if ( v5 == 3 )
15       return 1;
16   }
17   return result;
18 }
```

Figure 20. *Referer_check_or_common_cmd function body.*

be correct and we can successfully end this function (Figure 21).

Figure 22 shows two major checks related to the referrer. At line 16, instead of strictly checking for a referrer value like memcp or strcmp, developers have used the strstr function. The creators likely assumed it's enough just to check if the referrer contains string "127.0.0.1", which in their opinion, means that the request is coming from localhost.

The referrer string might contain the entire URL of a web page the request is coming from, which means it could hold the string "127.0.0.1" at any part and the check will be passed. There are even Referrer-Policy settings, which we can modify using meta tags.

We can check that behavior by choosing one of the config values to read without authentication, but with the correct referrer string set (Figure 23).

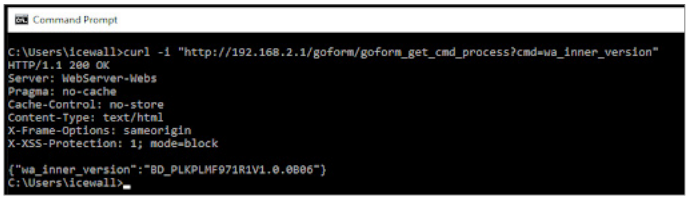

Figure 21. *An example of config value "wa_inner_version" we can get access to without any verification (authorization/ referrer check).*

```
1  int __fastcall referer_check(websRec *web)
2  {
3    int referer; // r0
4    int schema_len; // r7
5    char_t *referer_; // r5
6    char host[88]; // [sp+10h] [bp-58h] BYREF
7
8    memset(host, 0, 0x40u);
9    referer = web->referer;                    // Referer header
10   if ( (web->flags & WEBS_SECURE) != 0 ) //http or https ?
11     schema_len = 8;
12   else
13     schema_len = 7;
14   if ( referer )
15   {
16     if ( strstr(referer, "127.0.0.1") )
17     {
18       zte_syslog_append(6, 377417, 0xDF0, 0, "Referer is 127.0.0.1");
19     }
20     else
21     {
22       snprintf(host, 0x40u, "%s/", web->host);
23       if ( strstr(referer, host) != referer + schema_len )
24       {
25         zte_syslog_append( "is_Refer_right referer = [%s], host=[%s]",
26                            referer,
27                            web->host);
28         return 0;
29       }
30     }
31     status = 1;
32   }
33   return status;
34 }
```

Figure 22. *Referer_check function body.*

As we assumed, an attacker can create a file with a name containing the string "127.0.0.1" hosted at any domain and can bypass this referer check in a context of CSRF protection.

This vulnerability is CVE-2021-21745 – our full chain exploit.

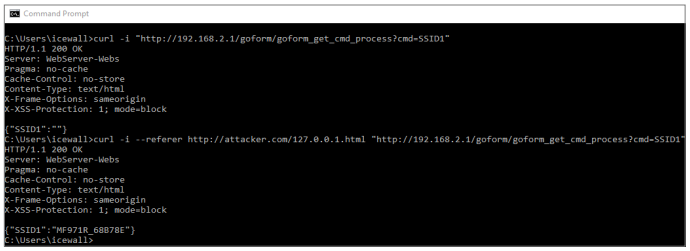

Figure 23. *Sending request for SSID1 value with and without referrer set.*

WebToken

We need to add an additional requirement more when we decide to communicate with the **"/goform/goform_set_cmd_process"** endpoint (Figure 24).

There is a function called at "AD_hash_check_and_ allowedlist", and Figure 25 shows its functionality.

The passed parameter called "AD" at line 7 is checked with some calculated value based on the `RD` value read from the config at line 20 and lines 17 - 41. Generally speaking, it's additional CSRF protection added to MF971R when compared to the MF910 where to most of "goformId" APIs web application adds or needs to add the unique URL "AD" value calculated based on the previously obtained "RD" value.

There are two example requests in Figures 26 and 27.

Depending on how strong the randomness of the RD value is, we could try to brute-force it, but we won't focus on that here. Using pre-auth XSS, we can obtain RD value sending the request in Figure 27 and calculate the AD value.

This function will return true also when the "goformId" value is one of the entries in the "g_set_commands_v1" array (Figure 28).

If we choose one of these IDs, we don't need to care about the "AD" parameter in our request. After examining all related handlers for the above form IDs, something interesting stood out.

```
int __fastcall handler_goform_set_cmd_process(websRec *web, int a2, int a3)
{
  char *formID; // r5
  const char *formId; // r6
  int allowed_list_flag; // r6
  int index; // r6
  char loggedin_flag[20]; // [sp+8h] [bp-90h] BYREF
  char out_buffer[20]; // [sp+1Ch] [bp-7Ch] BYREF
  char product_type[104]; // [sp+30h] [bp-68h] BYREF

  memset(product_type, 0, 0x40u);
  memset(loggedin_flag, 0, sizeof(loggedin_flag));
  loggedIn_and_ipCheck(web, loggedin_flag);
  formID = (char *)get_value_of_param(web, "goformId", (int)&g_default_value);
  zte_syslog_append(6, 386609, 588, 0, "zte_goform_set_cmd_process -> goformId:[%s].", formID);
  if ( g_web_is_support_token && !AD_hash_check_and_allowedlist(web, byte_125F61, 0)
    || referer_check_or_common_cmd(web, &g_default_value) )
  {
    goto FAIL;
  }

  if ( strcmp("ok", loggedin_flag) )
  {                                     // NOT LOGGED IN !!!
    memset(out_buffer, 0, sizeof(out_buffer));
    zte_nvconfig_read("loginfo", out_buffer, 20);
    formId = (const char *)get_value_of_param(web, "goformId", (int)&g_default_value);
    if ( !formId )
      zte_syslog_append(6, 386609, 783, 0, "zte_goform_whitelist_check: zte_goform_id is null.");
    zte_syslog_append(6, 386609, 785, 0, "zte_goform_set_cmd_process -> goformId:[%s].", formId);
    if ( !strcmp("SET_WEB_LANGUAGE", formId)
      || !strcmp("LOGIN", formId)
      || !strcmp("ADB_MODE_SWITCH", formId)
      || !strcmp("REDIRECT_REDIRECT_OFF", formId)
      || !strcmp("SET_MATCHED_LANGUAGE_FLAG", formId)
      || !strcmp("SET_REMIND_FLAG", formId)
      || !strcmp("DATA_USAGE_NOTIFY_RESET", formId)
      || !strcmp("ONLINE_UPGRADE_TOOL", formId)
      || !strcmp("SET_DEVICE_MODE", formId)
      || !strcmp("GOFORM_HTTPSHARE_CHECK_FILE", formId)
      || !strcmp("HTTPSHARE_ENTERFOLD", formId)
      || !strcmp("HTTPSHARE_FILE_RENAME", formId)
      || !strcmp("HTTPSHARE_NEW", formId)
      || !strcmp("HTTPSHARE_DEL", formId)
      || !strcmp("LIMIT_SPEED_REDIRECT_URL_CLEAR", formId)
      || !strcmp("SPEED_RESTRICTION_REDIRECT_SET", formId)
      || !strcmp("SPEED_RESTRICTION_STATE_SET", formId)
      || !strcmp("SET_TINYPROXY_SERVER_CONFIG", formId)
      || !strcmp("ALERT_WIFI_DFS_SET", formId) )
    {
      zte_syslog_append(6, 386609, 828, 0, "zte_goform_whitelist_check: zte_goform_id is OK.");
      allowed_list_flag = 1;
```

Figure 24. *Part of* ***goform_set_cmd_process*** *endpoint handler code.*

```
int __fastcall AD_hash_check_and_allowedlist(websRec *web, char *a2, int max_range)
{

  memset(RD, 0, sizeof(RD));
  memset(v16, 0, sizeof(v16));
  memset(s1, 0, 0x12Cu);
  AD_param = get_value_of_param(web, "AD", 0x5C86B);
  goformId = get_value_of_param(_web, "goformId", 0x5C86B);

  offset = 0;
  while ( strcmp(&g_set_commands_v1[offset], goformId) )
  {
    offset += 50;
    if ( offset == 200 )
    {
      v11 = *(unsigned __int8 *)AD_param;
      if ( *AD_param )
      {
        sub_530B0();
        zte_nvconfig_read("RD", RD, 64);
        snprintf(v16, 300u, "%s%s", RD_current_value, RD);
        v13 = RD_hash_calculation(v16);
        if ( !strcmp(v13, AD_param) )
        {
          v14 = 1;
        }
        else
        {
          v14 = 0;
          zte_syslog_append(6, 0x6EF23, 2704, 0, "zte_goform_accessid_check: Invaild accessid .");
        }
        sub_5317C();
        bfreeSafe(0x6EF23, 2710, (int)v13);
        result = v14;
      }
      else
      {
        zte_syslog_append(6, 0x6EF23, 2682, v11, "zte_goform_accessid_check: no accessid .");
        result = v11;
      }
      return result;
    }
  }

  return 1;
}
```

Figure 25. *AD_hash_check_and_allowedlist function body.*

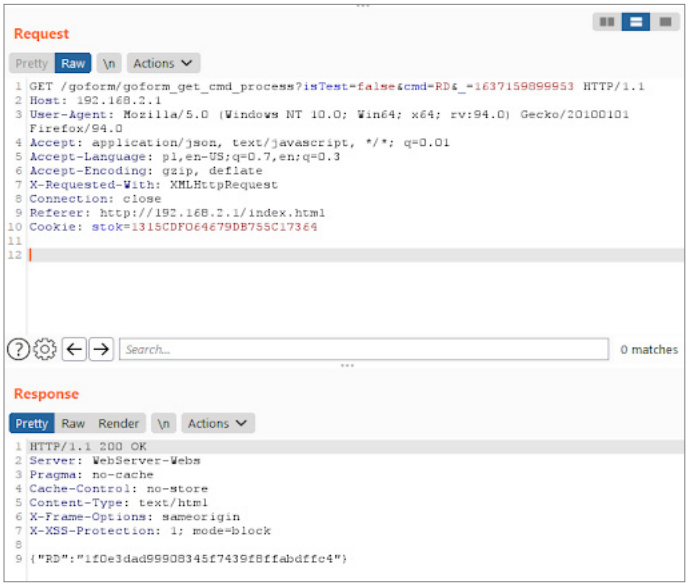

Figure 26. *Request sent to obtain the RD value.*

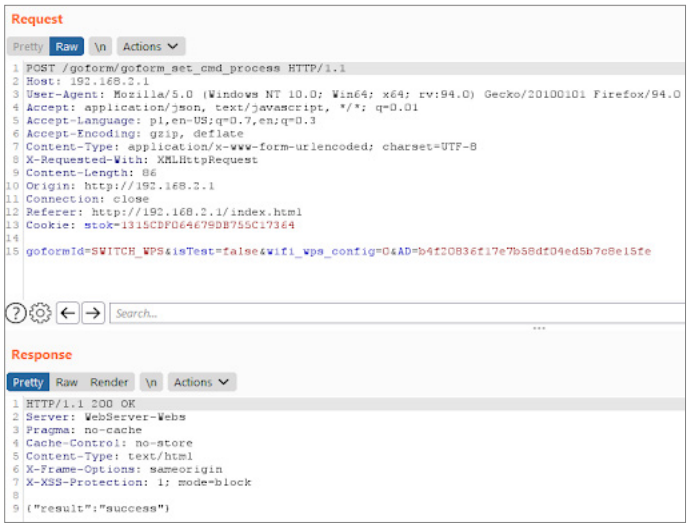

Figure 27. *AD value sent with a request to change WPS settings.*

```
g_set_commands_v1
=================
SET_WEB_LANGUAGE
LOGIN
ADB_MODE_SWITCH
ONLINE_UPGRADE_TOOL
```

Figure 28. *List of formIds not requiring prepare AD parameter.*

ADB_MODE_SWITCH

Before we move further, it's good to go back to the code presented in Figure 24 and note that, beside the fact that **"ADB_MODE_SWITCH"** form ID exists on a small list inside the **`AD_hash_check_and_allowedlist`** function called by me **"g_set_commands_v1"**.

It also exists on a type of "allowed list" in lines 30 - 48. To send requests to forms from that list, we don't need to be logged in. The only mitigation/requirement we need to pass is CSRF mitigation in a form of referrer check, which we already know how to bypass, even without XSS.

Armed with this knowledge, we'll look at the **"ADB_MODE_SWITCH"** handler (Figure 29).

Finally, we find a function where we can turn the ADB on line 22.

But it seems that we need to pass a proper password in lines 11 and 15.

Following the password parameter further, we land inside the **auth_mac** function of the **libzteencrypt.so** library (Figure 30).

Before we look at the **my_string_to_hex** function, remember the definition of **`hex_password`** buffer at line 8 and the fact that it can only contain around 1,024 characters (Figure 31).

The fully controlled data passed in the **"password"** parameter to **"my_string_to_hex"** is treated as ASCII hex (two input bytes are converted into one output byte) in line 18 and copied after that encoding into the **"hex_password"** buffer. There is no check against buffer overflow. It seems that we found a vulnerable function that is reachable without authentication and can be called via CSRF with a simple trick added to bypass referer mitigation. This is a perfect candidate for the pre-auth exploit we were looking for (Figure 32).

EXPLOITATION

ANY BINARY EXPLOITATION MITIGATION?

Now, it's time to check eventual exploitation mitigations we need to bypass to turn this stack-based buffer overflow into remote code execution (Figure 33).

There is no ASLR, relocations or stack canaries. Turning this stack BO into arbitrary code execution should be straightforward.

```
1  int __fastcall handler_ADB_MODE_SWITCH(websRec *web)
2  {
3    const char *password; // r5
4    char *v4; // r1
5    websRec *v5; // r0
6    char cmd[272]; // [sp+8h] [bp-110h] BYREF
7
8    memset(cmd, 0, 0x100u);
9    if ( !web )
10     return zte_syslog_append(6, 454435, 5302, 0, "wp is null.");
11   password = (const char *)get_value_of_param(web, "password", 378987);
12   zte_syslog_append(6, 454435, 5307, 0, "zte_device_adb_mode_switch:[password] is [%s].", password);
13   if ( *password )
14   {
15     if ( auth_mac((int)password) )
16     {
17       zte_syslog_append(6, 454435, 5317, 0, "auth failed");
18     }
19     else
20     {
21       memset(cmd, 0, 0x100u);
22       strcpy(cmd, "echo 1 >  /sys/class/android_usb/android0/debug_enable");
23       if ( j_system(cmd) != -1 )
24       {
25         zte_syslog_append(6, 454435, 5333, 0, "zte_device_adb_mode_switch:  set adb mode SUCCESS.");
26         v4 = "success";
27         v5 = web;
28         return responseError(v5, v4);
29       }
30       zte_syslog_append(6, 454435, 5328, 0, "zte_device_adb_mode_switch:  system command ERROR.");
31     }
32   }
33   v4 = "failure";
34   v5 = web;
35   return responseError(v5, v4);
36 }
```

Figure 29. *ADB_MODE_SWITCH function body.*

```
1  int __fastcall handler_ADB_MODE_SWITCH(websRec *web)
2  {
3    const char *password; // r5
4    char *v4; // r1
5    websRec *v5; // r0
6    char cmd[272]; // [sp+8h] [bp-110h] BYREF
7
8    memset(cmd, 0, 0x100u);
9    if ( !web )
10     return zte_syslog_append(6, 454435, 5302, 0, "wp is null.");
11   password = (const char *)get_value_of_param(web, "password", 378987);
12   zte_syslog_append(6, 454435, 5307, 0, "zte_device_adb_mode_switch:[password] is [%s].", password);
13   if ( *password )
14   {
15     if ( auth_mac((int)password) )
16     {
17       zte_syslog_append(6, 454435, 5317, 0, "auth failed");
18     }
19     else
20     {
21       memset(cmd, 0, 0x100u);
22       strcpy(cmd, "echo 1 >  /sys/class/android_usb/android0/debug_enable");
23       if ( j_system(cmd) != -1 )
24       {
25         zte_syslog_append(6, 454435, 5333, 0, "zte_device_adb_mode_switch:  set adb mode SUCCESS.");
26         v4 = "success";
27         v5 = web;
28         return responseError(v5, v4);
29       }
30       zte_syslog_append(6, 454435, 5328, 0, "zte_device_adb_mode_switch:  system command ERROR.");
31     }
32   }
33   v4 = "failure";
34   v5 = web;
35   return responseError(v5, v4);
36 }
```

Figure 30. *auth_mac function code.*

```
1  size_t __fastcall my_string_to_hex(const char *password, char *hex_password, size_t *out_size)
2  {
3    signed int offset; // r4
4    size_t result; // r0
5    signed int strlen_2; // r5
6    int v9; // [sp+0h] [bp-20h] BYREF
7    int byte; // [sp+4h] [bp-1Ch] BYREF
8
9    offset = 0;
10   v9 = 0;
11   byte = 0;
12   result = strlen(password);
13   strlen_2 = result >> 1;
14   *out_size = result >> 1;
15   while ( offset < strlen_2 )
16   {
17     strncpy((char *)&v9, &password[2 * offset], 2u);
18     result = sscanf((const char *)&v9, "%02X", &byte);
19     hex_password[offset++] = byte;
20   }
21   return result;
22 }
```

Figure 31. *my_string_to_hex function code.*

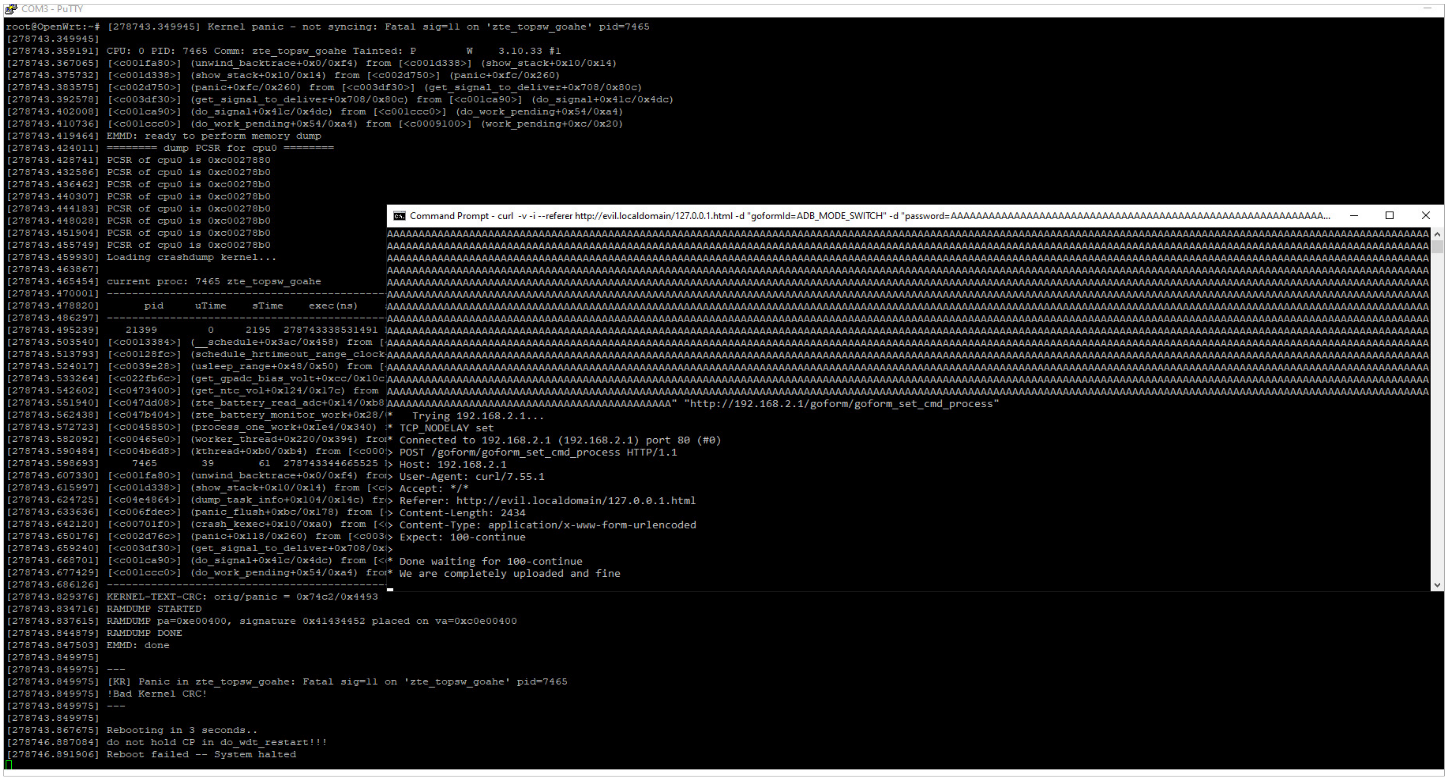

Figure 32. *Successful attempt of triggering stack based buffer overflow in ADB_MODE_SWITCH handler.*

```
icewall@ubuntu:~/tools/checksec.sh-2.5.0$ ./checksec --file=/tmp/zte_topsw_goahead
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
No RELRO        No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols        No    0               0               /tmp/zte_topsw_goahead
```

Figure 33. *Checksec output for zte_topsw_goahead binary.*

DEBUGGING SETUP

Even if our mission seems to look quite easy, we still need to have a possibility to debug our target. There is a great GitHub repository where we can find statically compiled tools for the ARM platform, including gdb and gdbserver.

My first approach was to use GDB directly on the device. That worked, but in a very limited way. I had some problems with stability and I couldn't use GEF to simplify any tasks – that's why I ended up with gdbserver running on the ZTE device attached to the zte_topsw_goahead process (Figure 34).

Because I was using Windows as my host machine, I decided to use **WSL** and install **gdb-multiarch + gef**. Then, I could set a breakpoint at the end of the **"auth_mac"** function to observe the stack BO and obtain necessary information (Figure 35).

RET2CODE

Since **"zte_topsw_goahead"** doesn't support ASLR and we can fully control data used to overflow the buffer (we can even pass 00, which will be converted into a null byte), I decided the easiest way to exploit this is to call the **"system"** function with passed **cmd** to execute in payload. So our plan to potentially exploit this is to:

- If, necessary use small ROP.
- Find a location where $r0 will be set to $SP+/- X.
- Depending on the $SP+/- X value, put the "hex encoded" cmd for system() in the payload at the proper offset.
- Find a location in the zte_topsw_goahead code where the "system" function is used.

I managed to manually find a perfect place in the code meeting all of these requirements (Figure 36).

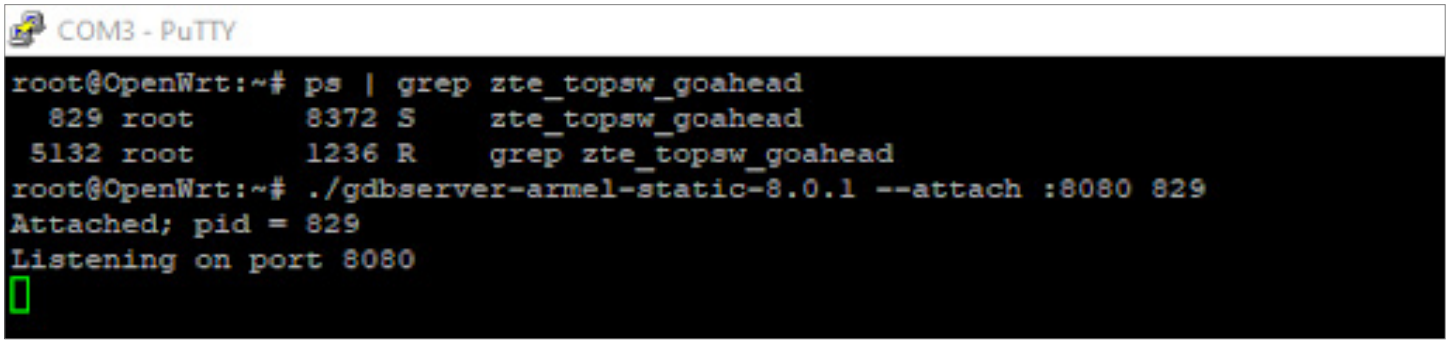

Figure 34. *Attach gdbserver to zte_topsw_goahead binary.*

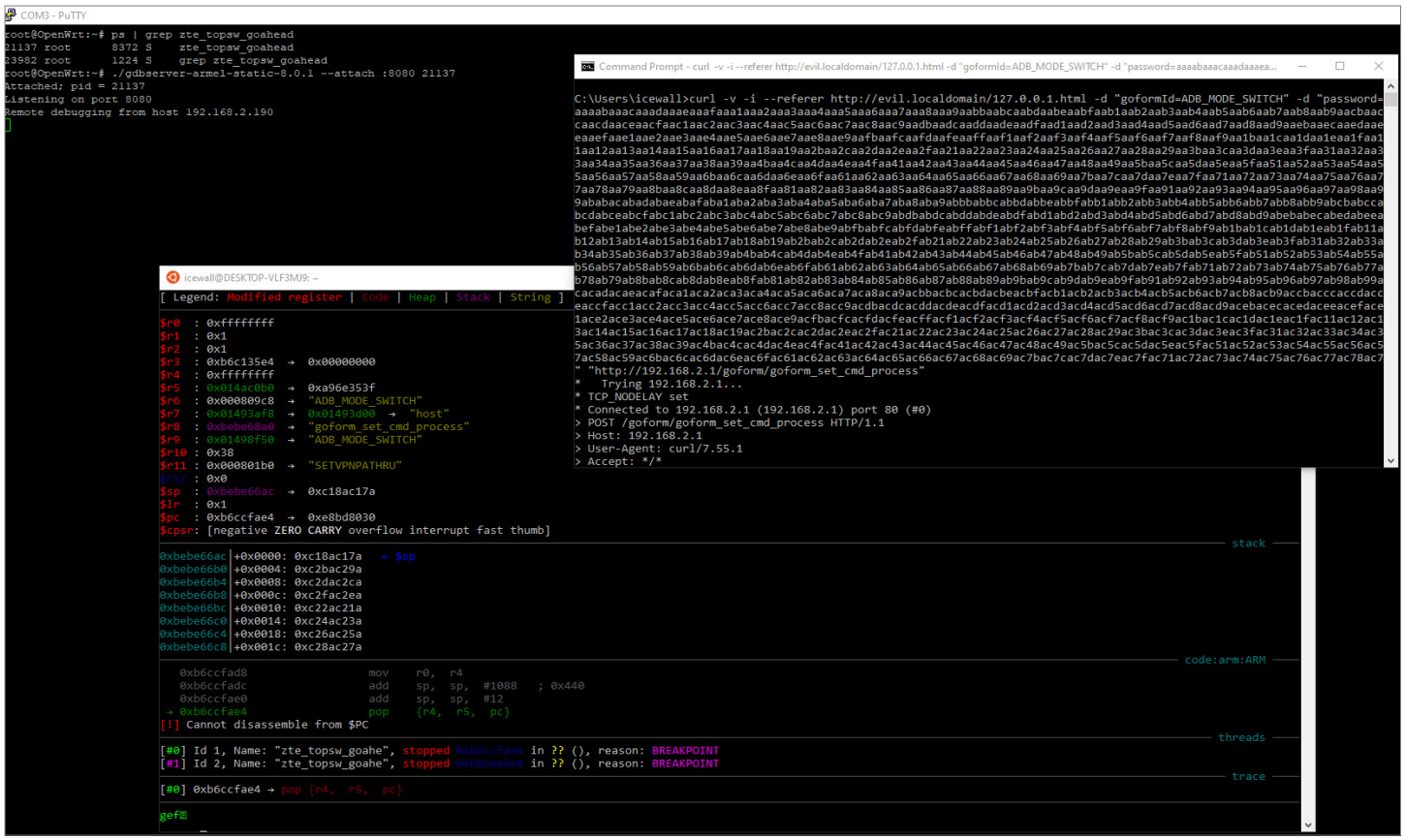

Figure 35. *Putty(UART <-> ZTE connection) and WSL with gdb-multiarch and GEF.*

```
LOAD:0003EBF4                ADD     R0, SP, #0x54 ; 'T' ; command
LOAD:0003EBF8                BL      system
```

Figure 36. *Location used to execute arbitrary cmd in our exploit.*

Now we can redirect code to set up $r0 with $SP and call the system function with our cmd string. We just need to put our cmd at the proper offset.

Our final layout of the **"password"** buffer is shown in Figure 37.

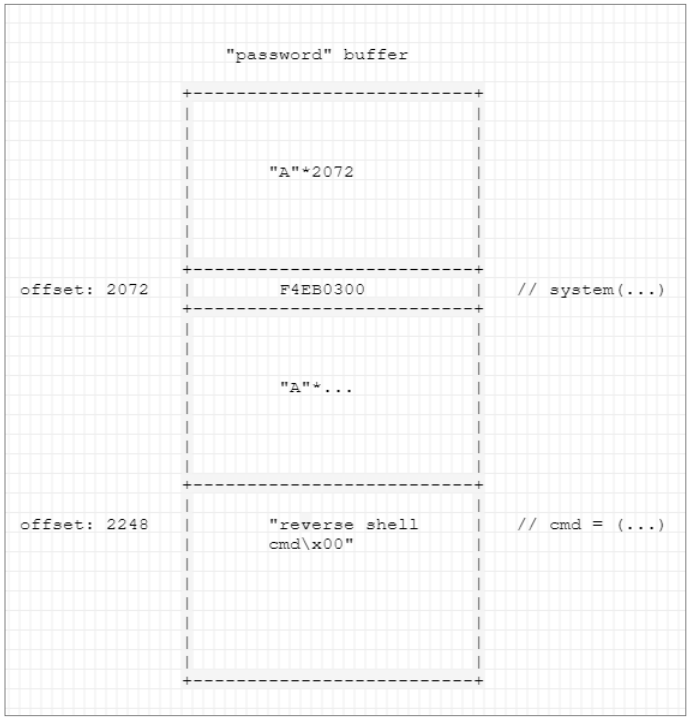

Figure 37. *"password" buffer layout.*

FINAL SHAPE OF THE EXPLOIT

Now, when we have all necessary information to put together the final exploit. For a video walkthrough of how this works, click here or visit cs.co/ZTEvideo.

We will:

- Create a HTML file with a name : 127.0.0.1.html to bypass CSRF protection.
- Use meta "referrer" with "content" set to "unsafe-url" to force the browser to send the full URL in the "referer" header.
- Create an HTML form where the value of the "password" field will be set to our payload.
- Return address and cmd, which will be treated as an ASCII hex, so we need to properly encode them in the payload (Figure 38).
- For demo purposes, users will need to click a button to trigger the exploit. Normally, after visiting this malicious website, a form would be auto-submitted.

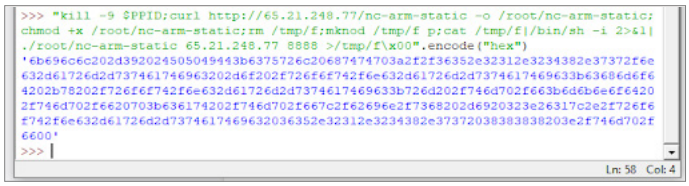

Figure 38. cmd *used to obtain the reverse shell on the device.*

SUMMARY

At the beginning of this research, I wasn't sure if achieving pre-auth remote code execution on MF971R device would be possible, due the fact of previously conducted research related with the ZTE MF series and additional mitigations introduced in the most recent version. Nevertheless, it turned out that precise and deep analysis allowed us to obtain the necessary pieces to execute remote arbitrary commands on the device with the highest privileges and minimal user interaction.

In a real-world scenario, attackers could use these vulnerabilities to infect random ZTE MF routers and create a botnet or precisely attack a particular person knowing they use such a model. Having a root access to that crucial device as a network router an attacker could monitor/ modify the victim's network traffic but also conduct attacks on devices available only in the internal network.Talos will continue to discover and responsibly disclose vulnerabilities on a regular basis and provide additional deep-dive analysis when necessary. Check out our original disclosure here to find out how you can keep your system protected from this vulnerability.