layout	tags	title
post	system hardware reverse crypto	Reversing TL-WR840N

Buying cheap routers to find vulnerabilities in them

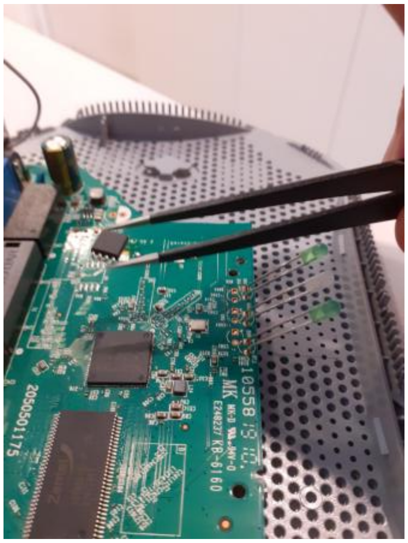
system hardware reverse crypto
`{:refdef: style="text-align: center;"} ![_config.yml]({{ site.baseurl }}/images/tplink/router.jpg) {: refdef}`

Part 1: Acquiring the firmware

TP Link firmwares are easy to get as they are free to download online: https://www.tp-link.com/in/support/download/tl-wr840n/ But that is a bit too easy isn't it ? Is there another way to get it ?

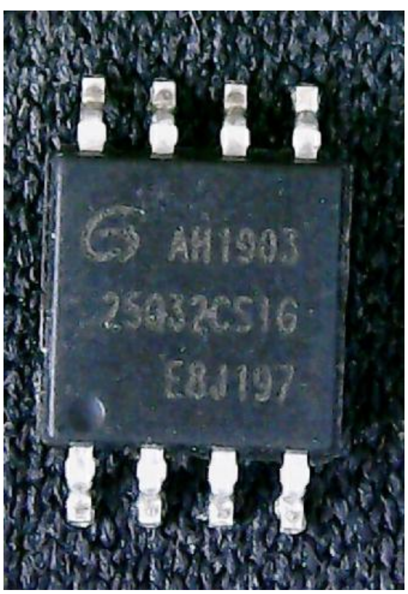
Most of these devices have a serial port for debugging. Connecting to it should be trivial using a USB to UART adapter:

Using this, you could potentially interrupt the boot sequence and enter a low-level shell that would allow you to do things like read the EEPROM chip at different offsets. Of course, if the goal is to easily get the firmware (over several hours because of the baud rate), then you've won ! But, still no thanks. I wanted to try something new:



Desoldering the EEPROM that contains the firmware was my way. If you are looking for a good reason to chose this way rather than any of the aformentioned ones, there isn't. I had just recently acquired a TL866II+, and wanted to have fun with it ! So the chip perfectly fits in the 200mm SOP8 adapter:

But before being able to read from it, we need to identify which kind of chip it is:



We can read AH1903 25Q32CS1G, and the logo is Giga Device. By looking for GD25Q32 in the list of devices supported by minipro, we get this: GD25Q32 @SOP8. We then dump the firmware using the Linux fork of the minipro tool, available here: https://gitlab.com/DavidGriffith/minipro/ using the command line

```
root@kali:~# minipro -p GD25Q32 -r firmware.bin
Found TL866II+ 04.2.86 (0x256)
Warning: Firmware is out of date.
```

```
  Expected  04.2.118 (0x276)
  Found     04.2.86 (0x256)
Chip ID OK: 0xC84016
Reading Code...  6.99Sec  OK
```

Works like a charm!

Part 2: Reversing the firmware

The file is 4Mb large, and contains the following signatures:

```
root@kali:~# binwalk firmware.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
53488         0xD0F0          U-Boot version string, "U-Boot 1.1.3 (Jun 14 2018 - 11:06:28)"
66048         0x10200         LZMA compressed data, properties: 0x5D, dictionary size: 8388608 bytes, uncompressed size: 2986732 byte
1048576       0x100000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2955905 bytes, 610 inodes, block
4063248       0x3E0010        XML document, version: "1.0"
```

We are mostly interested in the **squashfs**, as it contains the router's filesystem. But, we also can see that the bootloader code starts at 0xD0F0, so there may be something interesting stored at the beginning of the firmware.
All the binaries are compiled for **MIPS**, that is the most common architecture for routers. The /etc folder contains several interesting files. Notably, two encrypted XML files, *default_config.xml* and *reduced_data_model.xml*. By searching for these filenames inside the whole squashfs, we get one interesting match: **libcmm.so**.
Compiled code referencing encrypted files might be actually decrypting them, so we disassemble the shared object library using Ghidra.

```
int dm_decryptFile(uint param_1,undefined4 param_2,uint param_3,int param_4)

{
  int iVar1;
  char acStack40 [8];
  int local_20;

  memcpy(acStack40,&encryption_key,8);
  if (param_3 < param_1) {
    cdbg_printf(8,"dm_decryptFile",0xb83,
                "Buffer exceeded, decrypt buf size is %u, but dm file size is %u",param_3,param_1);
    local_20 = 0;
  }
  else {
    local_20 = cen_desMinDo(param_2,param_1,param_4,param_3,acStack40,0);
    iVar1 = local_20;
    if (local_20 == 0) {
      cdbg_printf(8,"dm_decryptFile",0xb8a,"DES decrypt error\n");
    }
    else {
      do {
        local_20 = iVar1;
        if (((undefined *)(param_4 + local_20))[-1] != '\0') break;
        iVar1 = local_20 + -1;
      } while (local_20 != 0);
      *(undefined *)(param_4 + local_20) = 0;
    }
  }
  return local_20;
}
```

If that ain't luck...
We have all the symbols! This makes the task much easier. So I renamed the *&encryption_key* pointer, but you can easily see that the code is:

1. Copying an 8 bytes array in a local buffer (DES key sizes are 7+1 for parity)
2. Calling *cen_desMinDo* which comes from **libcutil.so**, and is a wrapper for DES encryption functions

So we quickly get to the results using:

```
openssl enc -d -des-ecb -nopad -K XXXXXXXXXXXXXXXX -in default_config.xml > default_config_decrypted.xml
```

Of course, in this XML file, we expect to find loot, like:

```
    <StorageService>
    <UserAccount instance=1 >
    <Enable val=1 />
    <Username val=admin />
    <Password val=admin />
    <X_TP_Reference val=0 />
    <X_TP_SupperUser val=1 />
    </UserAccount>
```

We like those default credentials a lot, but they are usually widely known and/or documented (especially for cheap routers), so nothing very shocking here. The other file, reduced_data_model_decrypted.xml, contains the same kind of credentials and information, but this time they are models for the global configuration scheme.

Another intersting thing in /etc folder: **shadow doesn't exist**

The passwords are stored the old way, in the passwd file:

```
admin:$1$$iC.dUsGpxNNJGeOm1dFio/:0:0:root:/:/bin/sh
dropbear:x:500:500:dropbear:/var/dropbear:/bin/sh
nobody:*:0:0:nobody:/:/bin/sh
```

So *admin* seems to have a $1 (MD5) password. Let's see what hashcat thinks about this:

```
root@kali:~# hashcat -a 0 -m 500 hash /usr/share/wordlists/rockyou.txt

$1$$iC.dUsGpxNNJGeOm1dFio/:1234

Session..........: hashcat
Status...........: Cracked
Hash.Name........: md5crypt, MD5 (Unix), Cisco-IOS $1$ (MD5)
Hash.Target......: $1$$iC.dUsGpxNNJGeOm1dFio/
Time.Started.....: Wed Oct 28 14:10:47 2020 (1 sec)
Time.Estimated...: Wed Oct 28 14:10:48 2020 (0 secs)
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     3279 H/s (11.38ms) @ Accel:256 Loops:125 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests
Progress.........: 3072/14344385 (0.02%)
Rejected.........: 0/3072 (0.00%)
Restore.Point....: 0/14344385 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:875-1000
Candidates.#1....: 123456 -> dangerous
```

admin:1234 is the root's password of the router. You could use these credentials to connect through the serial port for example. Of course, we were not really expecting complicated passwords as defaults credentials.
You could also get the default credentials by visiting https://www.tp-link.com/us/support/faq/191
But why doing things simply.

Part 3: Dropbear

Another interesting user here is dropbear. It is a lightweight ssh server for embedded systems, and it could allow a user to authenticate remotely to the router. But how are the dropbear credentials initialized ?

By looking for dropbear related data, we stumble, once again, upon **libcmm.so**, and the very intriguing string `/var/tmp/dropbear/dropbearpwd`
The file doesn't exist in the squashfs, so it must be created during startup. In fact, it is created in one function in named **setDropbearLogin**:

```c
undefined4 setDropbearLogin(int conf_obj)

{
  undefined4 uVar1;
  FILE *__stream;
  size_t length;
  byte *pbVar2;
  int iVar3;
  char *__s;
  char acStack112 [36];
  byte dest [32];
  undefined local_2c;
  char *password;

  memset(dest,0,0x21);
  if ((*(char *)(conf_obj + 0x22) == '\0') || (*(char *)(conf_obj + 0x32) == '\0')) {
    cdbg_printf(8,"setDropbearLogin",0xe3,"uname = %s, pswd = %s\n",conf_obj + 0x22,conf_obj + 0x32)
    ;
    uVar1 = 1;
  }
  else {
    __stream = fopen("/var/tmp/dropbear/dropbearpwd","wb+");
    uVar1 = 1;
    if (__stream != (FILE *)0x0) {
      fprintf(__stream,"username:%s\n",conf_obj + 0x22);
      password = (char *)(conf_obj + 0x32);
      __s = acStack112;
      length = strlen(password);
      iVar3 = 0;
      cen_md5MakeDigest(dest,password,length);
      memset(acStack112,0,0x21);
      do {
        pbVar2 = dest + iVar3;
        iVar3 = iVar3 + 1;
        sprintf(__s,"%02x",(uint)*pbVar2);
        __s = __s + 2;
      } while (iVar3 != 0x10);
      memcpy(dest,acStack112,0x21);
      local_2c = 0;
      fprintf(__stream,"password:%s\n",dest);
      fclose(__stream);
      uVar1 = 0;
    }
  }
  return uVar1;
}
```

Hopefully, once again, Ghidra's output is very easily readable, but here is a summary:

1. Open and create /var/tmp/dropbear/dropbearpwd

2. Write the string "username:" with the username given as parameter
3. Hash the password given as parameter
4. Write the string "password:" with the 16 bytes hex encoded hash obtained previously
5. Close the dropbearpwd file

Note that the hashing function is an old friend as well, the function **cen_md5MakeDigest** also comes from **libcutil.so**. So now, how are the username and password given to our function? They actually come from a much larger structure.

The router uses a large object stored in the BSS. It is a global data model object, that is written using

```
undefined4 dm_setObj(uint param_1,ushort *param_2,ushort *param_3)
```

and read with

```
undefined4 dm_getObj(uint oid,ushort *out_buf,uint size,void *in_buf)
```

Integer OIDs are used to refer to specific parts of the configuration in the memory, for example, in our case, the OID for dropbear credentials is 8:

```
iVar1 = dm_getObj(8,&source,0x62,dropbear_obj_addr);
if (iVar1 != 0) {
  cdbg_printf(8,"prepareDropbear",0x11a,"get OID_USER_CFG error.\n");
}
setDropbearLogin(dropbear_obj_addr);
util_execSystem("prepareDropbear","dropbearkey -t rsa -f %s",
                "/var/tmp/dropbear/dropbear_rsa_host_key");
util_execSystem("prepareDropbear","dropbearkey -t dss -f %s",
                "/var/tmp/dropbear/dropbear_dss_host_key");
util_execSystem("prepareDropbear","dropbear -p %d -r %s -d %s -A %s",0x16,
                "/var/tmp/dropbear/dropbear_rsa_host_key",
                "/var/tmp/dropbear/dropbear_dss_host_key","/var/tmp/dropbear/dropbearpwd");
```

This code comes from the function that calls setDropbearLogin. It gets the *dropbear_obj_addr* from the global data model, at the OID 8. This section will contain both a username and a plaintext password.

Now, after tracing all the calls to *dm_setObj* (and it took a while):

```
**************************************************************
*                          FUNCTION                          *
**************************************************************
undefined dm_setObj()

dm_setObj                                   XREF[135]:   Entry Point(*),
                                                         rsl_sys_log:0002c470(c),
                                                         rsl_initPingWatchDogObj:0002e0f0
                                                         rsl_initL2tpConnPorttriggeringOb
                                                         rsl_initDmzHostCfgObj:0003c81c(c
```

...

It was clear the the OID 8 was never set as a hardcoded value, but actually from a loop iterating over values from a global variable named **configXMLCtx**. And by drawing the call graph to *dm_setObj* with the *configXMLCtx* as OID parameter, we find: {:refdef: style="text-align: center;"} `![_config.yml]({{ site.baseurl }}/images/tplink/callgraph.png)` {: refdef}

Our credentials come from the XML file we decrypted earlier: *default_config.xml* !!

Part 4: Command injections

Regarding this part, I have no certainty towards the vulnerabilities (mostly because the router is dead from desoldering) and it would require a live check to see if parameters are indeed injectable. This all starts with the prominent use of **system**, that, as you know, is widely discouraged.

```
                **************************************************************
                *                       THUNK FUNCTION                       *
                **************************************************************
                thunk int system(char * __command)
                  Thunked-Function: <EXTERNAL>::system
                  assume t9 = 0xb8160
int               v0:4           <RETURN>
char *            a0:4           __command
                system                                     XREF[8]:     Entry Point(*),
                                                                        rsl_sys_restoreDefaultCfg:0002af
                                                                        rsl_sys_updateFirmware:0002b280(
                                                                        util_execSystem:00092aac(c),
                                                                        util_execSystem_long:00092d38(c),
                                                                        oal_startUPnP:00098f18(c),
                                                                        ipt_init:000a20e0(c),
                                                                        000ecda4(*)
```

But **system** itself is not the problem, it is the **util_execSystem** function that relies on it, that is **everywhere** in the code. It has **495** cross references, in functions like *delStaticRoute* or *oal_ping*.

The function starts like this:

```
memset(command_line,0,0x200);
iVar1 = vsnprintf(command_line,0x1ff,cmd,&local_res8);
cdbg_printf(8,"util_execSystem",0x8b,"%s cmd is \"%s\"\n",caller_name,command_line);
if (0 < iVar1) {
  iVar1 = 1;
  do {
```

```
        local_22c = system(command_line);
        local_22c._1_1_ = (byte)(local_22c >> 8);
        local_240 = local_22c & 0x7f;
        if ((int)local_22c < 0) {
          if (local_22c == 0xffffffff) {
            cdbg_printf(8,"util_execSystem",0x9b,"system fork failed.");
          }
          else {
            perror("util_execSystem call error:");
          }
        }
      }while (waitpid)
```

The buffer containing the command is zeroed, then snprintf'd from the command line passed as a parameter, and straight from there (no sanitization whatsoever) passed to *system()*.

Here is a good example of potentially injectable situation:

```
void oal_ipt_addBridgeIsolationRules(undefined4 param_1)

{
  util_execSystem("oal_ipt_addBridgeIsolationRules",
                  "iptables -t filter -I BRIDGE_ISOLATION -i br+ -o %s -j DROP",param_1);
```

param_1 could contain an IP address entered from the web interface. While being directly concatenated, it could contain executable code to gain root on the device (bind, reverse shell for example).

I didn't have time to explore all the different uses, but it would be very surprising if none of those were injectable from the web administration interface.

Part 5: Going further

There is no real conculsion to draw from this study. I bought this router mostly to have fun with it and that's exactly what I did. Passwords are weak, firmware is not encrypted, and so on and so on, but you get what you pay for.

Many things are left unchecked, and I may continue this post in a second part:

1. What is before the bootloader? (spart of it eems to contain hardware configuration, including serial port's configuration)
2. Kernel is 2.6, is there some TPLink proprietary additions to it?
3. Module tp_domain.ko tries to contact dead domains: tplinklogin.net and www.tplinklogin.net, why?
4. Find an exploitable command injection using *util_execSystem*

Resources

Recovering TPlink default config
TL866ii+

A little while after writing this article, I stumbled upon this one on Pierre Kim's blog, which shows very similar vulnerabilities. I suspect the code is significantly similar, and probably libraries even contain the exact same code.

We contacted TP Link and created the following CVE-2020-36178 related to the code injection issue. A specific mention to TP Link's security team is deserved, they are very serious and quick to respond, it is always satisfying to see security handled as a serious matter.

Enough for today!
Stay classy netsecurios.

Reverse Engineering Routers