Broken, Abandoned, and Forgotten Code, Part 7

Zachary Cutlip

In the previous post, I finished discussing the `abCheckBoardID()` function. I called attention to a checksum in the header generated by an unknown algorithm. I provided a python implementation of that algorithm ported from IDA disassembly. In total, I identified four fields parsed by t`his function, accounting for 30 bytes of the 58 byte header.

In this part I'll give an overview of the remaining functions that parse and validate the firmware header. By the end we will be able to generate a header that allows the firmware to be programmed to flash memory. I won't discuss each header field in quite as much detail as I did previously, but if you've made it this far, it shouldn't be too hard to understand how each field is used.

Updated Exploit Code

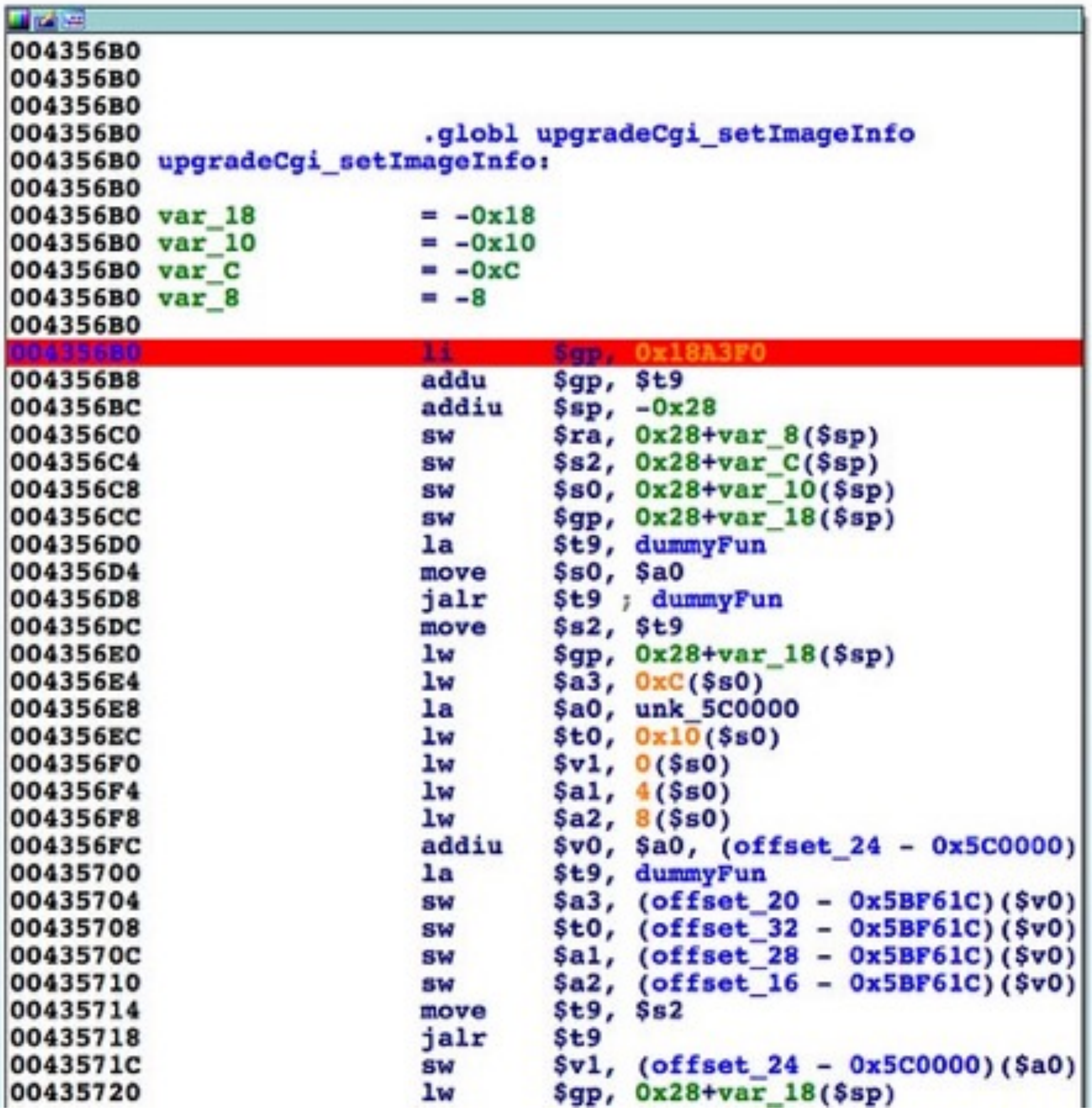
The update to the exploit code for Part 6 added a module to regenerate a checksum found in the header. This update populates a couple of additional checksums as well as a few other fields. The code provided for Part 7 is sufficient to generate a firmware header that will pass the web server's validation. Given a valid kernel and filesystem image, you should be able to generate a firmware image that the web interface will happily upgrade to. If you've previously cloned the repository, now would be a good time to do a pull. You can clone the git repo from: https://github.com/zcutlip/broken_abandoned

Of Checksums and Sizes

After the `abCheckBoardID()` function (discussed in part 6) there are a few more functions that parse or validate portions of the header.

Identifying these fields and their purpose is challenging due to the fact that values may be parsed out in one function, but not used until some other function or functions, if at all.

The two functions that parse out values from the header are `upgradeCgi_setImageInfo()` at 0x004356B0 and `upgradeCgiCheck()` at 0x004361F8. The "setImageInfo" function is a short one. It parses several header fields, but it doesn't inspect or use any of them. The values are stored in global variables for later use. You can identify offsets of these fields using string patterns as described previously. As you identify these locations where the parsed values are located, rename the variables in IDA to something more meaningful, so you can identify them later when they are used. I renamed them to correspond with the offsets they were parsed from.



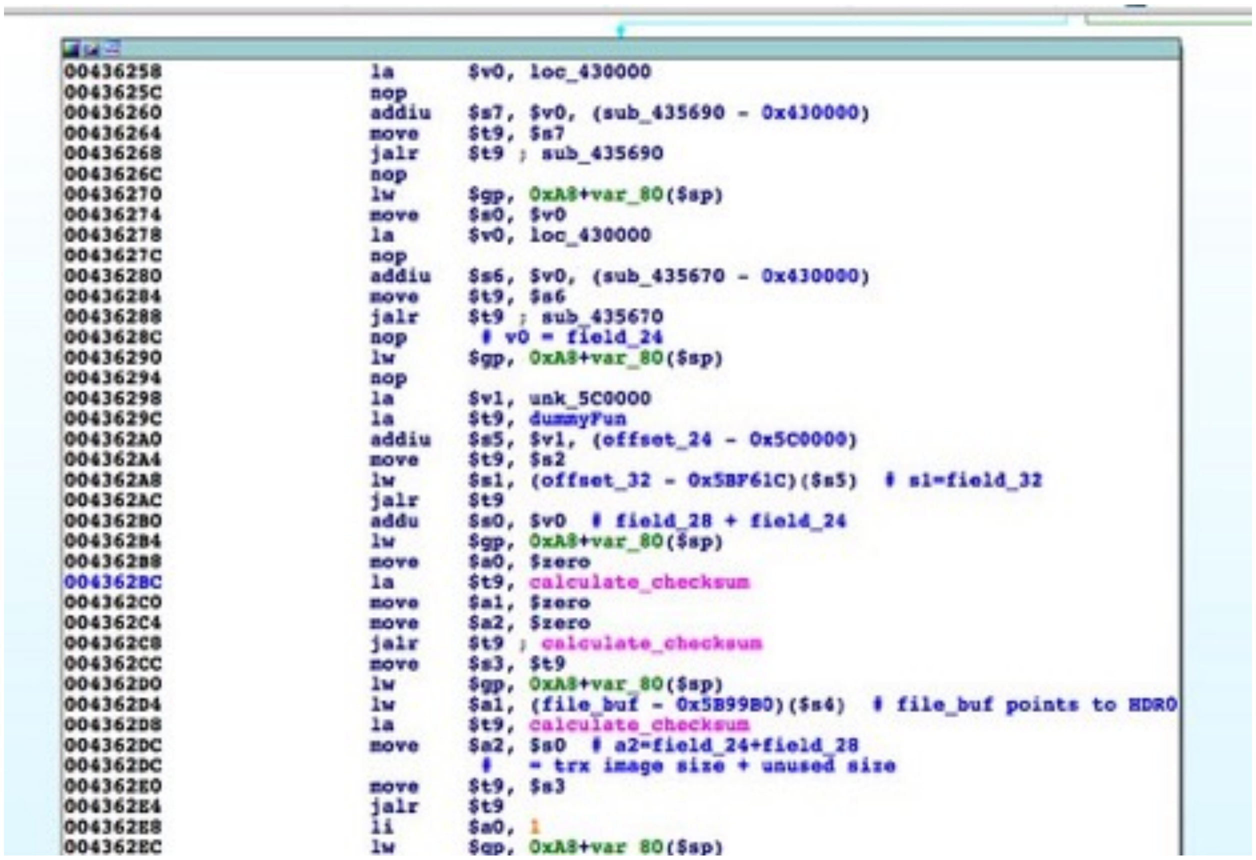
```
004356B0
004356B0
004356B0
004356B0                         .globl upgradeCgi_setImageInfo
004356B0 upgradeCgi_setImageInfo:
004356B0
004356B0 var_18                  = -0x18
004356B0 var_10                  = -0x10
004356B0 var_C                   = -0xC
004356B0 var_8                   = -8
004356B0
004356B0                         li      $gp, 0x18A3F0
004356B8                         addu    $gp, $t9
004356BC                         addiu   $sp, -0x28
004356C0                         sw      $ra, 0x28+var_8($sp)
004356C4                         sw      $s2, 0x28+var_C($sp)
004356C8                         sw      $s0, 0x28+var_10($sp)
004356CC                         sw      $gp, 0x28+var_18($sp)
004356D0                         la      $t9, dummyFun
004356D4                         move    $s0, $a0
004356D8                         jalr    $t9 ; dummyFun
004356DC                         move    $s2, $t9
004356E0                         lw      $gp, 0x28+var_18($sp)
004356E4                         lw      $a3, 0xC($s0)
004356E8                         la      $a0, unk_5C0000
004356EC                         lw      $t0, 0x10($s0)
004356F0                         lw      $v1, 0($s0)
004356F4                         lw      $a1, 4($s0)
004356F8                         lw      $a2, 8($s0)
004356FC                         addiu   $v0, $a0, (offset_24 - 0x5C0000)
00435700                         la      $t9, dummyFun
00435704                         sw      $a3, (offset_20 - 0x5BF61C)($v0)
00435708                         sw      $t0, (offset_32 - 0x5BF61C)($v0)
0043570C                         sw      $a1, (offset_28 - 0x5BF61C)($v0)
00435710                         sw      $a2, (offset_16 - 0x5BF61C)($v0)
00435714                         move    $t9, $s2
00435718                         jalr    $t9
0043571C                         sw      $v1, (offset_24 - 0x5C0000)($a0)
00435720                         lw      $gp, 0x28+var_18($sp)
```

Renaming global variables corresponding to header offsets.

The `upgradeCgiCheck()` function validates a few fields parsed out previously. At 0x004362BC we see the return of our friend, `calculate_checksum()`. This time the checksum is computed across more than just the firmware header. At the "update" step, the data argument points to the "`HDR0`" portion of the firmware. This suggests the checksum is across the TRX image that follows the 58 byte header. The size argument is the sum of the values found at offsets 24 and 28.

Inspecting the values at those positions in a stock firmware, we see 0x00871000 at offset 24, and 0x0 at offset 28. It's clear that bytes 24 - 27 are the size of the firmware image minus the 58 bytes at the start. Based on its use here, the bytes 28 - 31 are also a size of some sort.

At any rate, the size passed to `calculate_checksum()` at the update stage at 0x004362DC is the size of the TRX image. At 0x0043630C, the checksum is compared to the value taken from offset 32. We now know three more fields in the firmware header: offsets 24, 28, and 32. That's 42 bytes down, 16 to go.



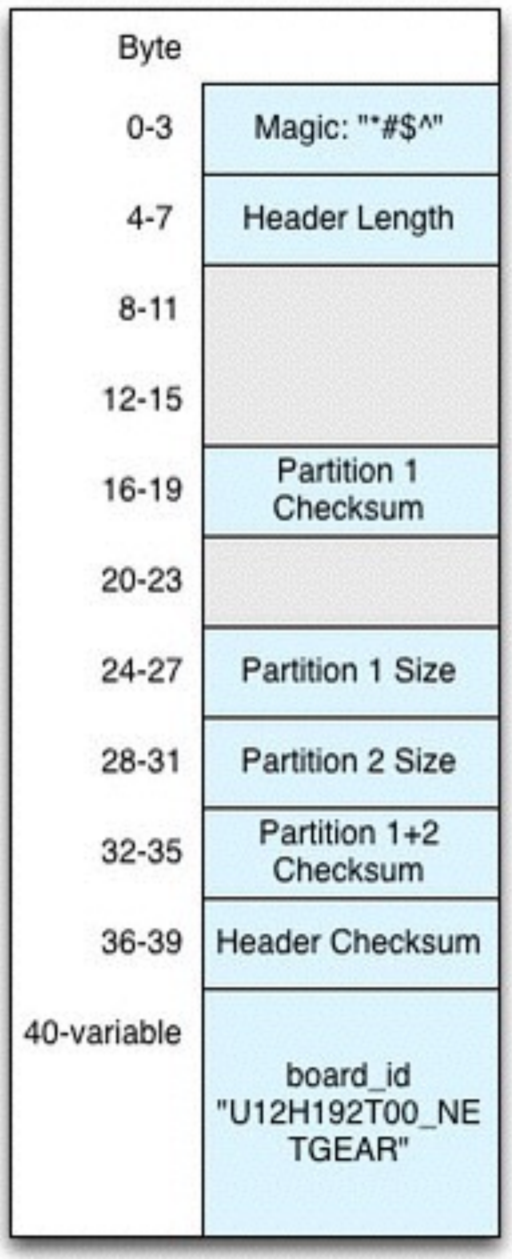
```
00436258                 la      $v0, loc_430000
0043625C                 nop
00436260                 addiu   $s7, $v0, (sub_435690 - 0x430000)
00436264                 move    $t9, $s7
00436268                 jalr    $t9 ; sub_435690
0043626C                 nop
00436270                 lw      $gp, 0xA8+var_80($sp)
00436274                 move    $s0, $v0
00436278                 la      $v0, loc_430000
0043627C                 nop
00436280                 addiu   $s6, $v0, (sub_435670 - 0x430000)
00436284                 move    $t9, $s6
00436288                 jalr    $t9 ; sub_435670
0043628C                 nop      # v0 = field_24
00436290                 lw      $gp, 0xA8+var_80($sp)
00436294                 nop
00436298                 la      $v1, unk_5C0000
0043629C                 la      $t9, dummyFun
004362A0                 addiu   $s5, $v1, (offset_24 - 0x5C0000)
004362A4                 move    $t9, $s2
004362A8                 lw      $s1, (offset_32 - 0x5BF61C)($s5)  # s1=field_32
004362AC                 jalr    $t9
004362B0                 addu    $s0, $v0  # field_28 + field_24
004362B4                 lw      $gp, 0xA8+var_80($sp)
004362B8                 move    $a0, $zero
004362BC                 la      $t9, calculate_checksum
004362C0                 move    $a1, $zero
004362C4                 move    $a2, $zero
004362C8                 jalr    $t9 ; calculate_checksum
004362CC                 move    $s3, $t9
004362D0                 lw      $gp, 0xA8+var_80($sp)
004362D4                 lw      $a1, (file_buf - 0x5B99B0)($s4)  # file_buf points to HDR0
004362D8                 la      $t9, calculate_checksum
004362DC                 move    $a2, $s0  # a2=field_24+field_28
004362DC                          #   = trx image size + unused size
004362E0                 move    $t9, $s3
004362E4                 jalr    $t9
004362E8                 li      $a0, 1
004362EC                 lw      $gp, 0xA8+var_80($sp)
```

Checksum of the firmware's TRX image.

We're not done with checksums just yet. The basic block at 0x0043643C is another checksum operation. Once again the data points to "`HDR0`", but the size is only the value from offset 24. The size from offset 28 is not used this time. The checksum result is the same as before, but this time compared to the value at offset 16. We now know the checksum we compute and store at offset 32 must also be stored at offset 16.

At this point we can speculate this firmware format supports multiple partitions or sections. The value at offset 24 would be the size of partition 1, and offset 28 would be the size of partition 2. The checksum at offset 16 would be calculated over partition 1, and offset 32's checksum would be calculated over partitions 1 and 2 combined.

We're now down to 12 unidentified bytes. Let's have a look at an updated header diagram to see how things look.



What we know so far about the firmware header.

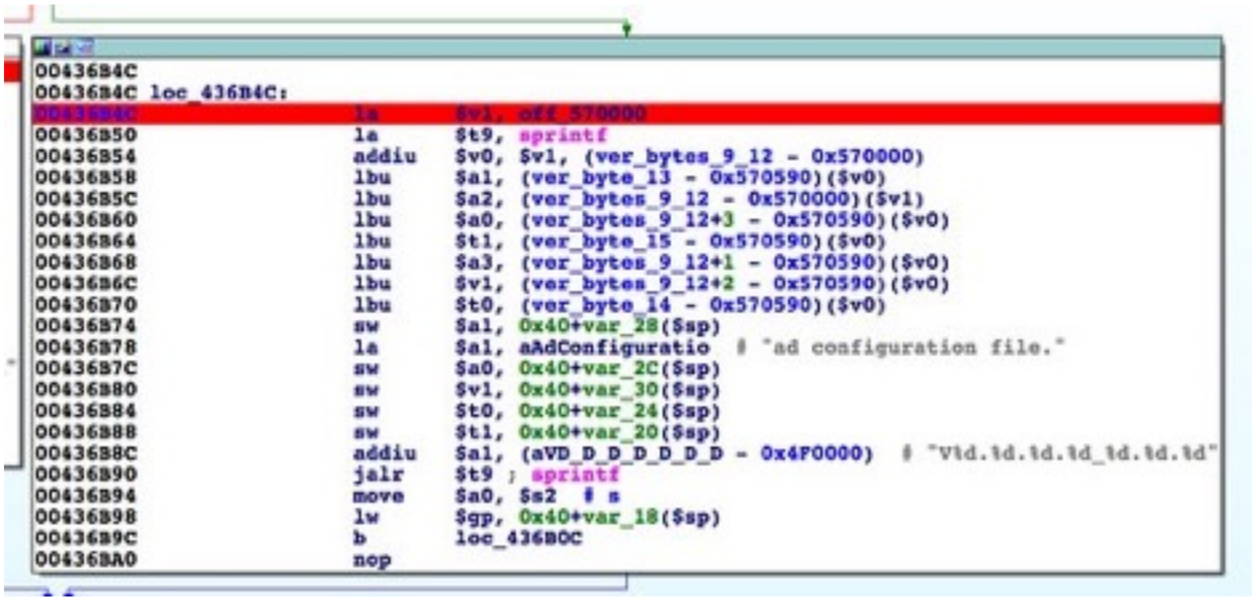
The diagram is starting to fill in, and things are looking quite a bit better.

Version String

Moving on, at 0x00436580, more data is parsed out of the firmware image. This time the values are pulled out one byte at a time. This frustrates the technique of using the 3+ byte patterns to identify offsets. Based on the format strings from subsequent `sscanf()` and `sprintf()` operations, we can speculate that these values are transformed in some way into the version string displayed in the web interface.

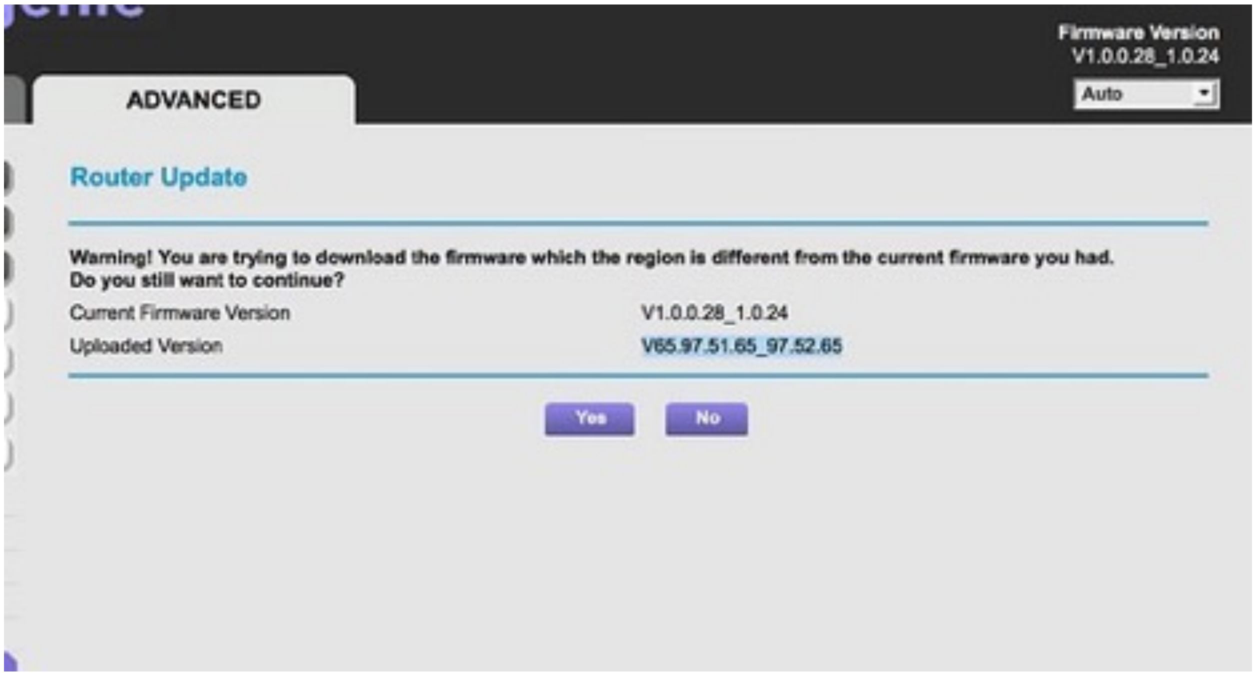
Although the version string ends up being only cosmetic, and not an essential part of the firmware validation, it's still interesting enough to discuss here. Modifying the version string would be a nice way to visually demonstrate that the target is, in fact, running your custom firmware, and not the stock firmware.
[**Update:** *Turns out this isn't quite right. There is a string table stored in flash memory that also contains the version string, and that string is displayed in the web interface. The version field in the firmware header is only (as far as I can tell) rendered during the update process so the user can see what version they're updating to*.]

It took some debugging, but it turns out the single byte values that compose the version string don't actually get used until a few functions later, in `upgradeCgi_GetParam()` at 0x00436B4C.



```
00436B4C
00436B4C loc_436B4C:
00436B4C                 la      $v1, off_570000
00436B50                 la      $t9, sprintf
00436B54                 addiu   $v0, $v1, (ver_bytes_9_12 - 0x570000)
00436B58                 lbu     $a1, (ver_byte_13 - 0x570590)($v0)
00436B5C                 lbu     $a2, (ver_bytes_9_12 - 0x570000)($v1)
00436B60                 lbu     $a0, (ver_bytes_9_12+3 - 0x570590)($v0)
00436B64                 lbu     $t1, (ver_byte_15 - 0x570590)($v0)
00436B68                 lbu     $a3, (ver_bytes_9_12+1 - 0x570590)($v0)
00436B6C                 lbu     $v1, (ver_bytes_9_12+2 - 0x570590)($v0)
00436B70                 lbu     $t0, (ver_byte_14 - 0x570590)($v0)
00436B74                 sw      $a1, 0x40+var_28($sp)
00436B78                 la      $a1, aAdConfiguratio  # "ad configuration file."
00436B7C                 sw      $a0, 0x40+var_2C($sp)
00436B80                 sw      $v1, 0x40+var_30($sp)
00436B84                 sw      $t0, 0x40+var_24($sp)
00436B88                 sw      $t1, 0x40+var_20($sp)
00436B8C                 addiu   $a1, (aVD_D_D_D_D_D_D - 0x4F0000)  # "V%d.%d.%d.%d_%d.%d.%d"
00436B90                 jalr    $t9 ; sprintf
00436B94                 move    $a0, $s2  # s
00436B98                 lw      $gp, 0x40+var_18($sp)
00436B9C                 b       loc_436B0C
00436BA0                 nop
```

What is happening here is a version string is being generated to display in the web browser so that the user can confirm what version of the firmware they're about to upgrade to.

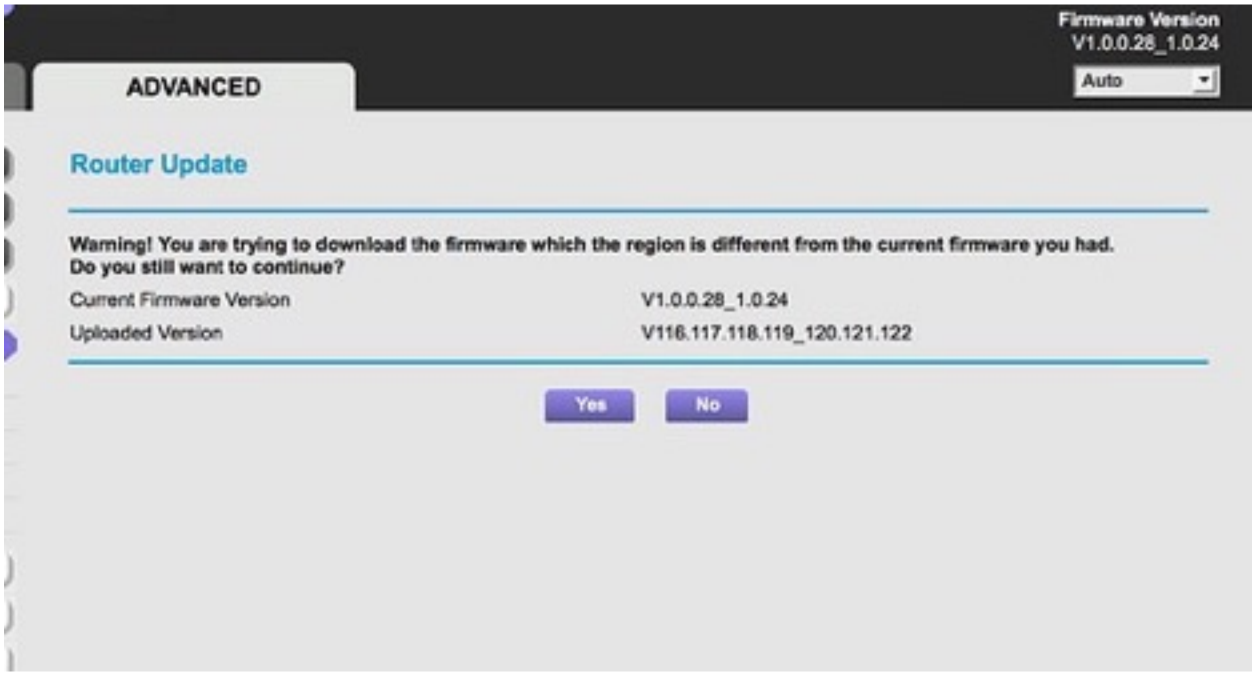


Router Update

Warning! You are trying to download the firmware which the region is different from the current firmware you had. Do you still want to continue?

Current Firmware Version	V1.0.0.28_1.0.24
Uploaded Version	V65.97.51.65_97.52.65

The version string "V65.97.51.65_97.52.65" from the screenshot above appears to be composed of the decimal representations of ASCII characters from Bowcaster's pattern string. We can be sure by replacing bytes 8 - 15 with a string of non-repeating characters: "stuvwxyz". When we do this, the version string becomes "V116.117.118.119_120.121.122". This confirms the hypothesis; these are the decimal representations for t,u,v,w,x,y, and z. Note that "s" is not included. Even though byte 8 was parsed out along with the rest, it appears to go unused.

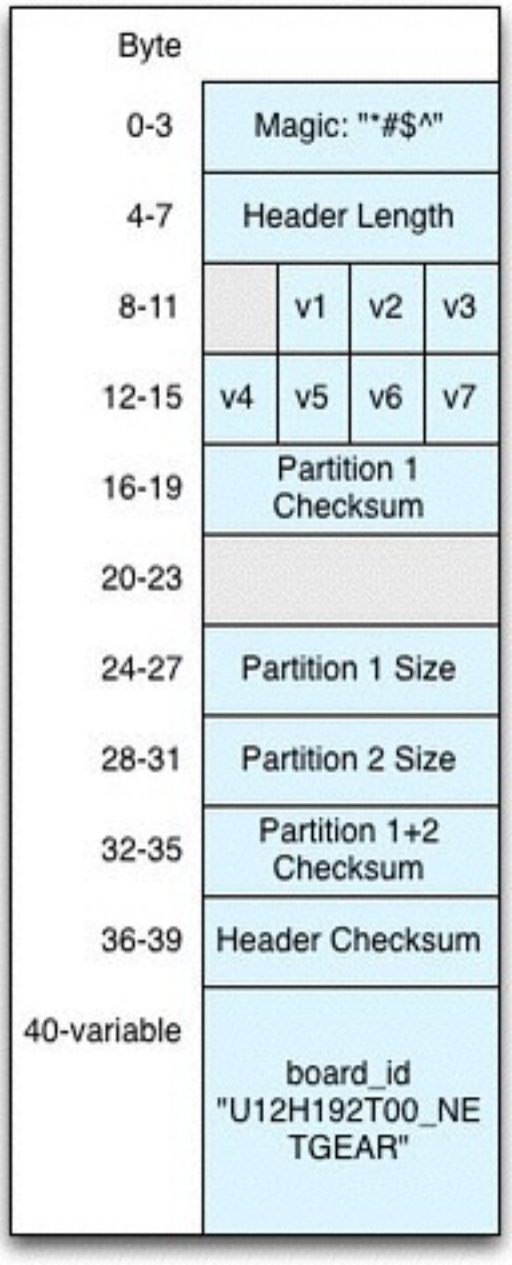


Router Update

Warning! You are trying to download the firmware which the region is different from the current firmware you had.
Do you still want to continue?

Current Firmware Version	V1.0.0.28_1.0.24
Uploaded Version	V116.117.118.119_120.121.122

We can now update the header diagram to reflect the version bytes.



(Mostly) Complete Firmware Header

The header diagram now has only 4 bytes (5 if you count the unused version byte at offset 8) that haven't been identified. It's unclear what these bytes are for, since they are never inspected. A likely explanation is that a checksum for theoretical partition 2 belongs at offset 20. The stock firmware has 0x0 at offset 20, which jives with a partition 2 size of 0. At any rate, this header is sufficient for execution to reach the point where the uploaded firmware gets written to `/dev/mtd1`.

WARNING: *If you are debugging httpd on on actual hardware rather than in emulation, there's a chance your router will end up bricked if you attempt to upgrade to a customer firmware image. Eventually, we must test on actual hardware, but before then, I'll describe how to access the device's serial console using a UART to USB cable. Using the serial console, you can recover from a bad firmware update, a feature I had to use many times during my original research.*

In the next part, with a better understanding of the firmware format, we'll loop back to the UPnP daemon and pick up where we left off there. Wouldn't it be nice if we could use the now documented header format to generate a firmware that will work with the UPnP daemon using our existing exploit code?