Broken, Abandoned, and Forgotten Code, Part 8

Zachary Cutlip

In the previous few posts, we spent time reversing how the Netgear R6200's HTTP daemon parses a firmware header before writing the firmware image to flash. The goal was to work out how the 58-byte firmware header is constructed and how to generate a new one that can replace the header in a stock firmware. In the end we identified the purpose of all but 4 bytes. The regenerated header plus the original TRX firmware image allowed the HTTP daemon, running in emulation, to reach the stage where it would start writing data to the `/dev/mtd1` flash partition. Considering this a win, we'll now circle back to analyzing `upnpd`.

In this and the next part, we'll compare the way `upnpd` parses and validates the firmware header to that of `httpd`. Having developed a baseline understanding of how the header is parsed by `httpd`, analyzing `upnpd` is much easier.

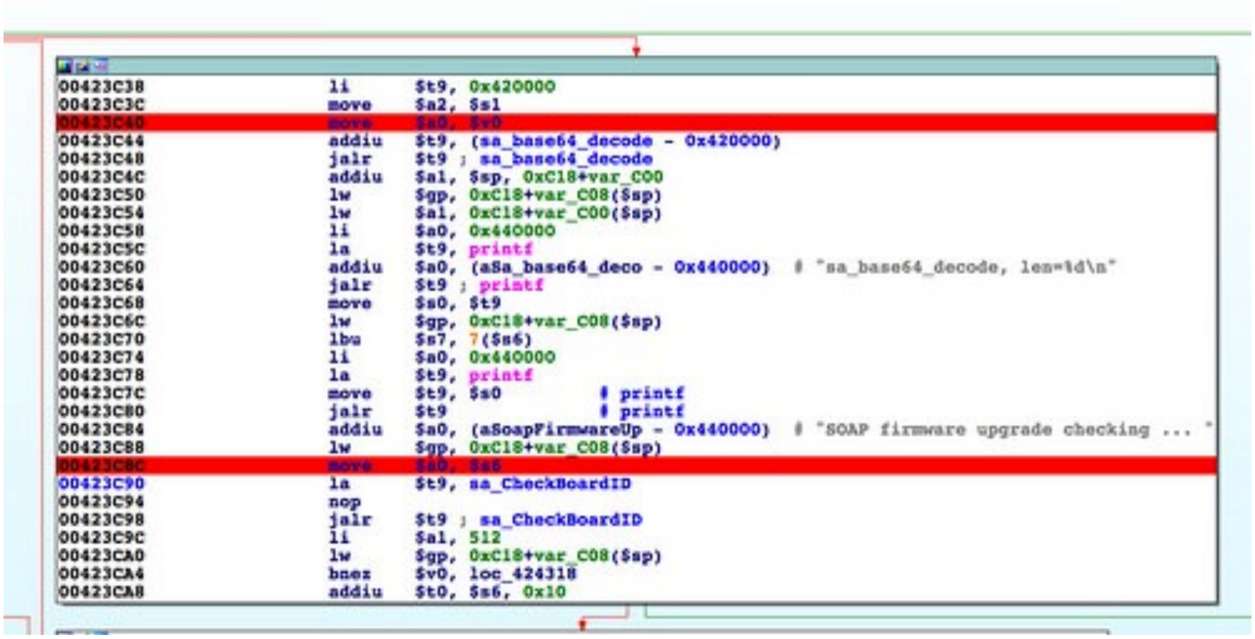
Updated Exploit Code

As in previous installments, the exploit code has been updated. Since we're switching back to `upnpd` in order to analyze how *it* validates the firmware, the repository contains separate modules for that. Look for `janky_ambit_header.py` and `build_janky_fw.py`. You can find the updated code and README in the `part_8` directory. Now is a good time to do a pull or to clone the repository from:
https://github.com/zcutlip/broken_abandoned

More Firmware Parsing, Pretty Much Like Before

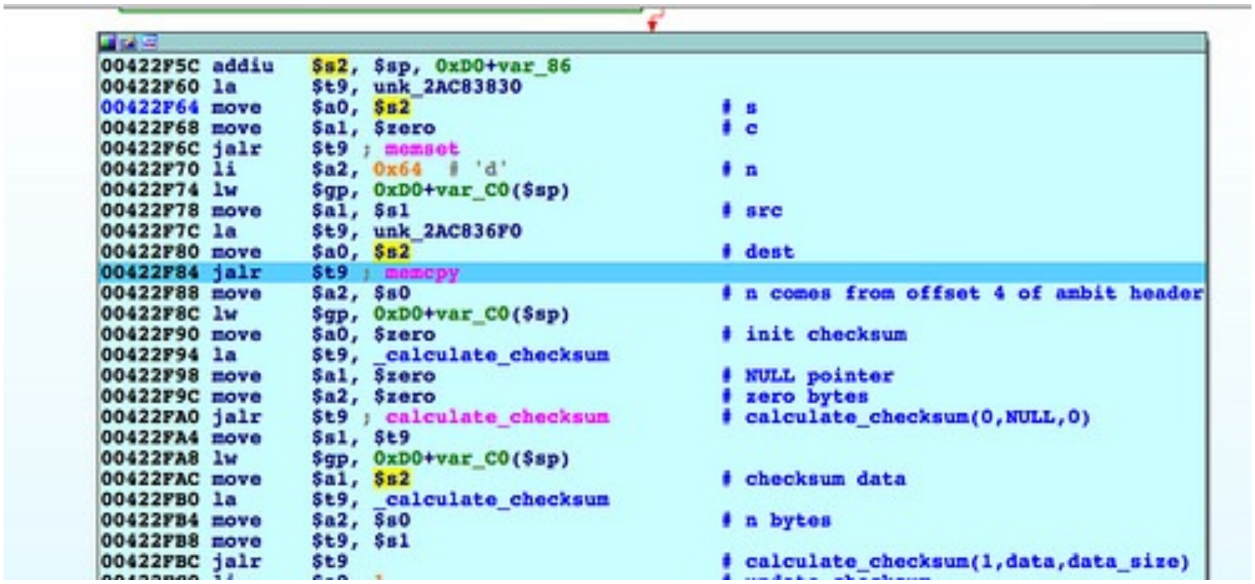
As we discovered in part 4, a firmware larger than 4MB will crash `upnpd` due to an undersized memory allocation. Obviously we won't be able to strap a header to the front of a stock TRX image like we did with `httpd`; it's way too big. Shrinking the firmware will be a challenge for later. If it turns out that we can't even get so far as writing the firmware to flash memory without crashing, it won't matter that you were able to shrink and re-pack the firmware. Instead, just dd out a little less than 4MB of random data from `/dev/random` and prepend a header to it. If you can get `upnpd` to write that image to flash, you win this stage and may advance to the next level.

Once we get past the undersized `malloc()` at 0x00423C24 in `sa_parseRcvCmd()`, the firmware is successfully base64 decoded out of the SOAP request. Then, at 0x00423C98, a function named `sa_CheckBoardID()` is called.



```
00423C38                li      $t9, 0x420000
00423C3C                move    $a2, $s1
00423C40                move    $a0, $v0
00423C44                addiu   $t9, (sa_base64_decode - 0x420000)
00423C48                jalr    $t9 ; sa_base64_decode
00423C4C                addiu   $a1, $sp, 0xC18+var_C00
00423C50                lw      $gp, 0xC18+var_C08($sp)
00423C54                lw      $a1, 0xC18+var_C00($sp)
00423C58                li      $a0, 0x440000
00423C5C                la      $t9, printf
00423C60                addiu   $a0, (aSa_base64_deco - 0x440000)  # "sa_base64_decode, len=%d\n"
00423C64                jalr    $t9 ; printf
00423C68                move    $s0, $t9
00423C6C                lw      $gp, 0xC18+var_C08($sp)
00423C70                lbu     $s7, 7($s6)
00423C74                li      $a0, 0x440000
00423C78                la      $t9, printf
00423C7C                move    $t9, $s0          # printf
00423C80                jalr    $t9               # printf
00423C84                addiu   $a0, (aSoapFirmwareUp - 0x440000)  # "SOAP firmware upgrade checking ... "
00423C88                lw      $gp, 0xC18+var_C08($sp)
00423C8C                move    $a0, $s6
00423C90                la      $t9, sa_CheckBoardID
00423C94                nop
00423C98                jalr    $t9 ; sa_CheckBoardID
00423C9C                li      $a1, 512
00423CA0                lw      $gp, 0xC18+var_C08($sp)
00423CA4                bnez    $v0, loc_424318
00423CA8                addiu   $t0, $s6, 0x10
```

This function should be familiar. It's nearly identical to the `abCheckBoardID()` function I described in part 5. So identical, in fact, that the buffer overflow via `memcpy()` I described previously is in this function as well.



```
00422F5C addiu   $s2, $sp, 0xD0+var_86
00422F60 la      $t9, unk_2AC83830
00422F64 move    $a0, $s2                             # s
00422F68 move    $a1, $zero                           # c
00422F6C jalr    $t9 ; memset
00422F70 li      $a2, 0x64  # 'd'                     # n
00422F74 lw      $gp, 0xD0+var_C0($sp)
00422F78 move    $a1, $s1                             # src
00422F7C la      $t9, unk_2AC836F0
00422F80 move    $a0, $s2                             # dest
00422F84 jalr    $t9 ; memcpy
00422F88 move    $a2, $s0                             # n comes from offset 4 of ambit header
00422F8C lw      $gp, 0xD0+var_C0($sp)
00422F90 move    $a0, $zero                           # init checksum
00422F94 la      $t9, _calculate_checksum
00422F98 move    $a1, $zero                           # NULL pointer
00422F9C move    $a2, $zero                           # zero bytes
00422FA0 jalr    $t9 ; calculate_checksum             # calculate_checksum(0,NULL,0)
00422FA4 move    $s1, $t9
00422FA8 lw      $gp, 0xD0+var_C0($sp)
00422FAC move    $a1, $s2                             # checksum data
00422FB0 la      $t9, _calculate_checksum
00422FB4 move    $a2, $s0                             # n bytes
00422FB8 move    $t9, $s1
00422FBC jalr    $t9                                  # calculate_checksum(1,data,data_size)
```

Buffer overflow due to memcpy() using header size field. Sad trombone.

Even the Buffer Overflow is the Same

To recap, the `memcpy()` is bounded only by the size value from the header. Since we control that value, we get precise control over how many bytes are copied into the destination buffer.

I didn't go into detail about the buffer overflow before, because I wanted to wait until I could discuss it in the context of `upnpd`. In the HTTP server, this isn't an interesting vulnerability. In that case, it is a post-authentication vulnerability. You would need to bypass authentication or trick a user into uploading your malicious firmware. If you've accomplished either of those, there are much more useful things you can be doing with your time than exploiting buffer overflows.

In the case of `upnpd`, this same vulnerability doesn't require authentication, making it much more interesting. Here's what's neat about it:

- No authentication required.
- The payload is base64 encoded and decoded for free, so there are no bad bytes to avoid related to the transport protocol.
- The buffer overflow is via `memcpy()` rather than a string handling function. There are no bad bytes to avoid related to string handling.
- The buffer being overflowed is on the stack, making it easy to overwrite the function's return address.

This is a straightforward buffer overflow. If you're new to stack based buffer overflows, or just new to exploiting memory corruption vulnerabilities on MIPS, this is an easy one to practice with, especially if you have the debugging environment I described here set up.

However, as I said in the first part of this series, one of my self-imposed goals was to avoid exploiting bugs along the way. We're trying to flash a firmware without crashing, and any bugs along the way are obstacles to overcome.

Working through this function reveals the same header fields that we discovered in its `httpd` counterpart: The magic number, the size and checksum of the header, and the board ID string. These fields are found at the same header offsets as before.

Mystery Header Gets a Name

There is one new piece of information, however.

```
00423078  # ---------------------------------------------------------------------------
00423078
00423078 loc_423078:                                  # CODE XREF: sa_CheckBoardID+A4'j
00423078                 li      $a0, 0x440000
0042307C                 la      $t9, puts
00423080                 nop
00423084                 jalr    $t9 ; puts
00423088                 addiu   $a0, (aNotAmbitImage_ - 0x440000)  # "Not Ambit image ... reject!!!"
0042308C                 lw      $gp, 0xD0+var_C0($sp)
00423090                 b       loc_423014
00423094                 li      $v0, 0xFFFFFFFF
00423094  # End of function sa_CheckBoardID
00423094
```

At 0x00423088 there is an error message that we didn't see in `httpd`: "Not Ambit image ... reject!!!". This is the first indication of any sort of name for this file format. This explains why you may have noticed references to "ambit" or "ambit header" in previous code fragments I've posted.

In the next part, we get close to writing the firmware image to flash memory. We'll have to do some binary patching to work around the fact that QEMU doesn't actually have flash memory.