Broken, Abandoned, and Forgotten Code, Part 6

Author: Zachary Cutlip

Note: *It is assumed that the reader is debugging the processes described in this and the next several posts using emulation and IDA Pro. Those topics are outside the scope of this series and are covered in detail here and here.*

In the previous post, we switched gears and started looking at the web server for the Netgear R6200. That's because the HTTP daemon's code for upgrading the firmware is less broken and easier to analyze. We also analyzed a stock firmware image downloaded from Netgear to see how it is composed. Craig Heffner's binwalk identified three parts, a TRX header at offset 58, followed by a compressed Linux kernel, followed by a squashfs filesystem. All of those parts are well understood, which only leaves the first 58 bytes to analyze.

With the goal of recreating the header using a stock TRX header, Linux kernel, and filesystem, I described how we can use Bowcaster to create fake header data to aid in debugging. When we left off, I had started discussing httpd's `abCheckBoardID()` function at 0x0041C3D8, which partially parses the firmware header. We identified a magic signature that should be at the firmware image's offset 0, as well as some sort of size field that should be at offset 4. We also discovered this header should be big endian encoded even though the target system is little endian.

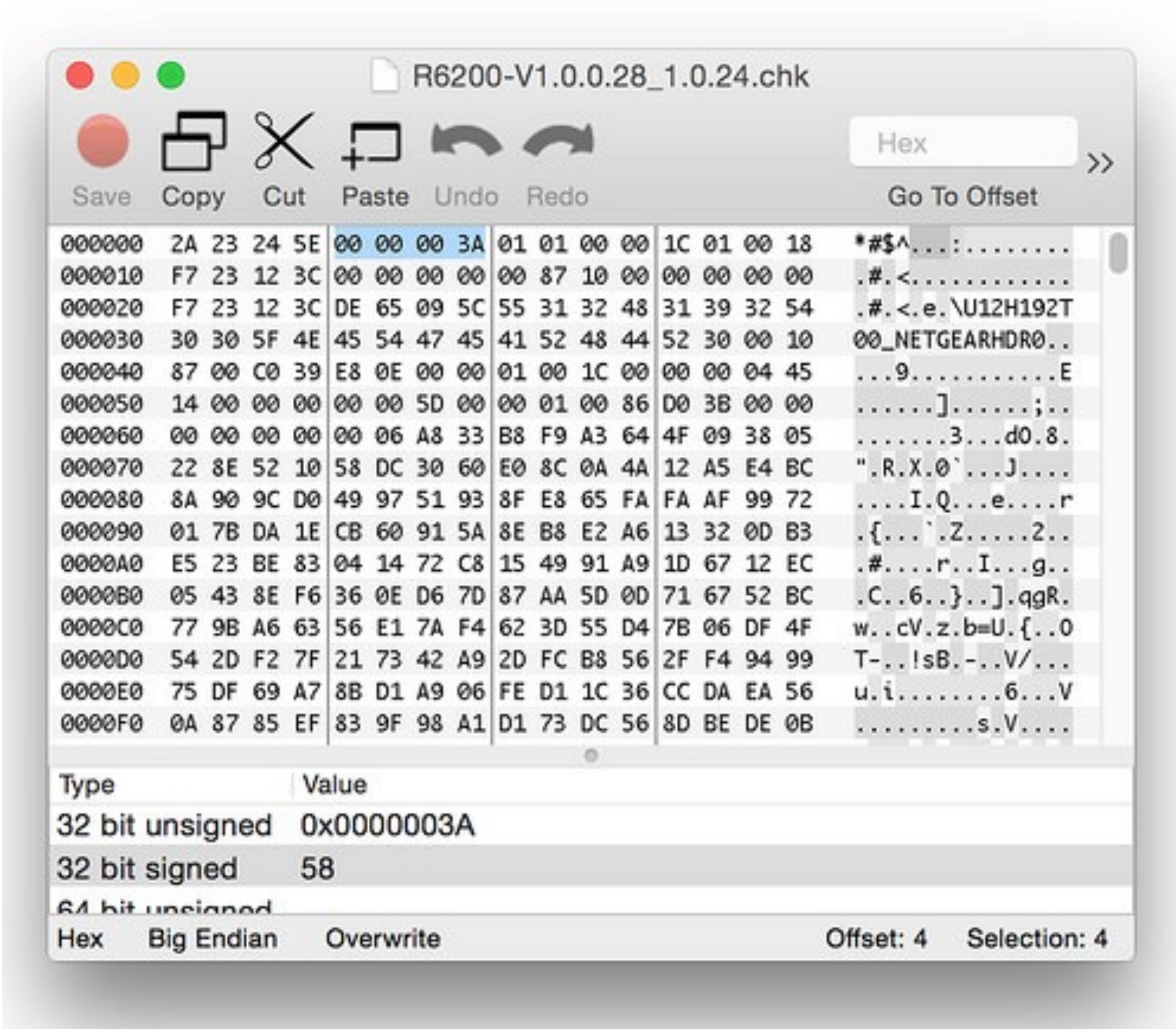
In this part, we'll clarify the purpose of the size field as well as identify a checksum field. Identification of the checksum algorithm is tricky if you don't have an eye for that sort of thing (I do not). I'll show how to deal with that. By the end of this part, we will have identified four fields, accounting for 30 bytes of the 58-byte firmware header.

Updated Exploit Code

I last updated the exploit code for part 5, which added several Python modules to aid in reverse engineering and reconstructing a firmware image. In this part I've added a module to regenerate checksums found in the header (see below). Additionally, the MysteryHeader class populates a couple of new fields that we will cover this post. 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

Header Size

We know the field at offset 4 is a size field of some sort because it's used as the size for a `memcpy()` operation[1]. Let's take a look at a stock firmware image to see what value is in that field. It might correlate to something obvious.

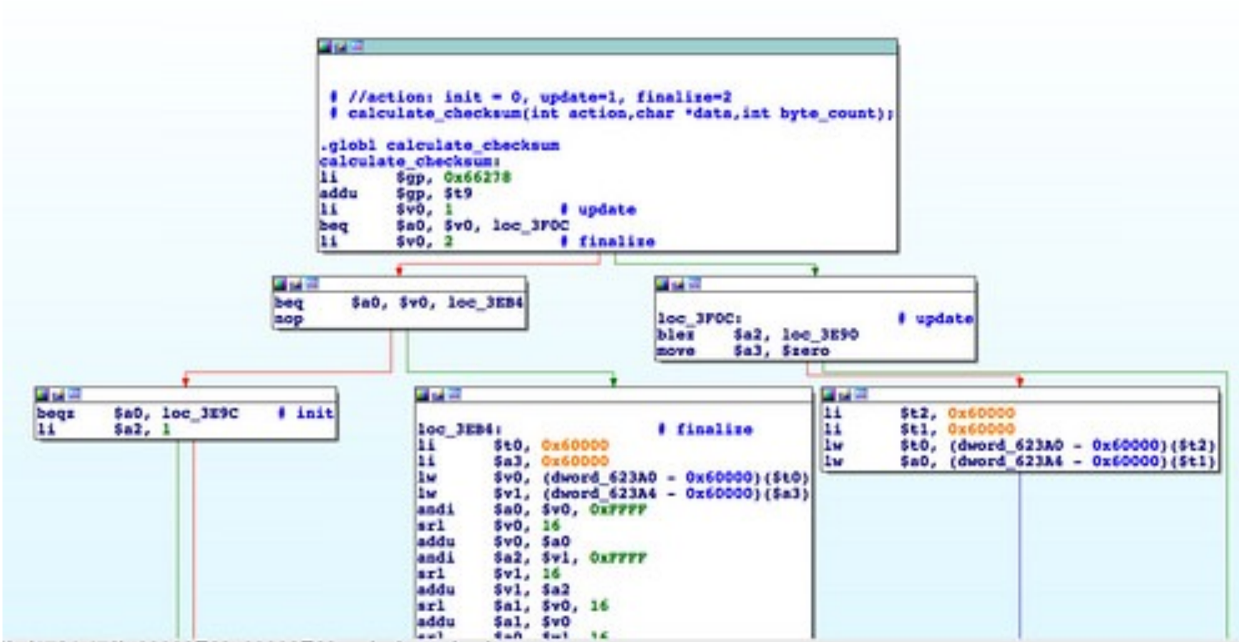


Above, we see the stock value is 0x0000003A, or 58 in decimal. Since 58 is also the amount of unidentified data before the TRX header, it's a safe bet this field is the overall size of this unidentified header. It's also a safe bet that this header is variable in size. The TRX header, whose size is fixed, does not have a size field for the header alone, only for the header plus data.

Checksumming the firmware header.

Checksum Fun

From `abCheckBoardID()` there are several calls to the `calculate_checksum()` function. This is an imported symbol and is not in the httpd binary itself. Strings analysis of libraries on the R6200's filesystem reveals that this function is in the shared library `libacos_shared.so`. We can disassemble this binary and analyze the function.



Disassembly of calculate_checksum().

There's no need to completely reverse engineer this function. Sure, it would be convenient to know what checksum algorithm this is[2] and if there was a built-in python module to use. All we really need, however, is code that calculates the same values this function does. It's easier in this case to just reimplement the algorithm. I duplicated this function one-for-one, where each line of MIPS disassembly became a line of Python. It's a small function, so it didn't take long to do. That module is included in this week's update to the git repo.

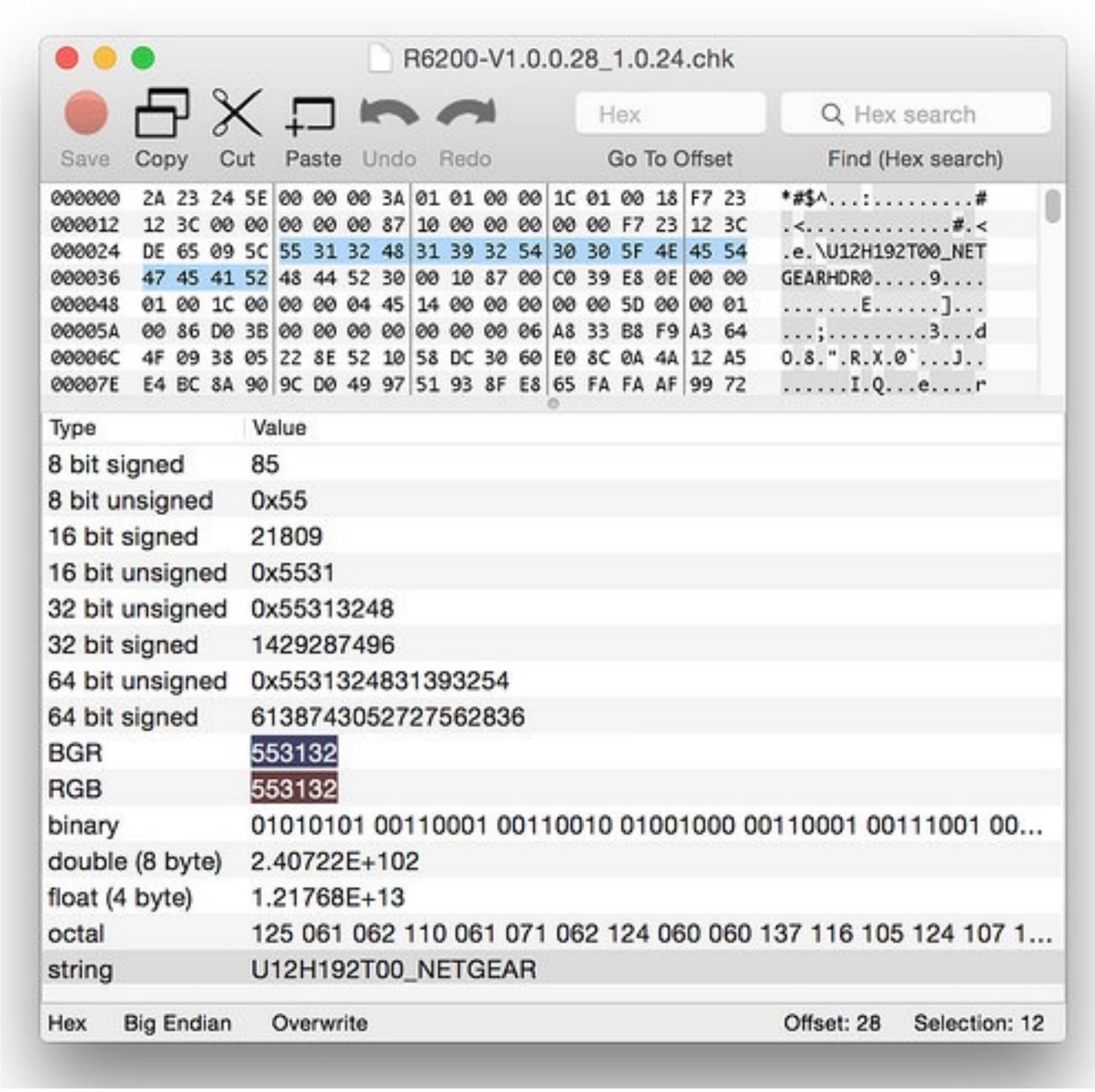
```
    def _update(self,data):
        size=len(data)
        t0=self.dword_623A0
        a0=self.dword_623A4
        a2=size
        a3=0
        while a3 != a2:
            v1=ord(data[a3])
            a3+=1
            a0=(a0+v1) & 0xffffffff
            t0=(t0+a0) & 0xffffffff

        self.dword_623A0=t0
        self.dword_623A4=a0

        return 1

    def _finalize(self):
        v0=self.dword_623A0
        v1=self.dword_623A4

        a0=(v0 & 0xffff)
        v0=(v0>>16 )
        v0=(v0+a0) & 0xffffffff
```

Python code fragment that looks suspiciously like IDA Pro disassembly.

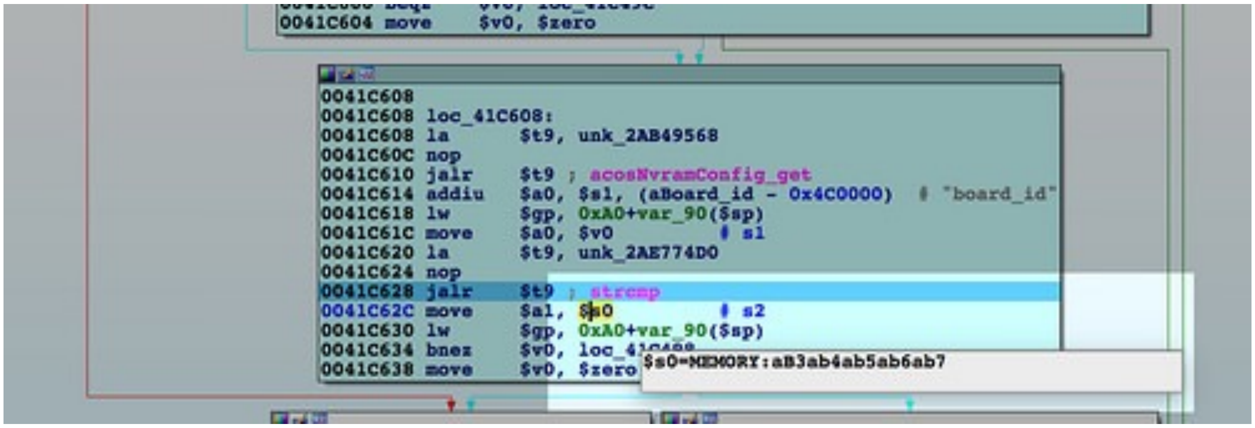
A checksum is calculated across the first 58 bytes of the header. Then at 0x0041C5BC the checksum gets compared to 0x41623241, a value extracted from the firmware data. Using Bowcaster's `find_offset()`, it is revealed that offset 36 of the firmware header should contain the checksum of the header itself. We'll need to calculate that value for the header and insert it at this location. In `abCheckBoardID()` the checksum field is zeroed out before the value is calculated. We should do the same before calculating our own. The updated code in the git repository performs this operation.

Board ID String

With the header checksum in place, we can move forward to the next few basic blocks. A few checks are performed to verify the "board_id" string of the firmware. There are a couple of hard-coded board_id strings that are referenced. If neither of those match, NVRAM is queried to find out the running device's board_id. It's possible to verify the proper board ID is "U12H192T00_NETGEAR" by extracting the NVRAM parameters from a live device[3]. Even if we didn't have that information, we could still analyze a stock firmware, where we find the same string embedded in the header.



As before, by looking at the pattern string that is compared, we can identify the offset into the header where the board_id should be placed.




```
$ ./buildfw.py find=b3Ab4Ab5Ab6Ab7Ab8A kernel.lzma
squashfs.bin
 [@] Building firmware from input files: ['kernel.lzma',
'squashfs.bin']
 [@] TRX crc32: 0x0ee839c0
 [@] Creating ambit header.
 [+] Building header without checksum.
 [+] Calculating header checksum.
 [@] Calculated header checksum: 0x840d0ddd
 [+] Building header with checksum.
 [@] Finding offset of b3Ab4Ab5Ab6Ab7Ab8A
 [+] Offset: 40
```

The string b3Ab4Ab5Ab6Ab7Ab8A is located at offset 40.

It is worth noting that we suspected the header was variable length given the presence of a size field. The board_id is a string and is the last field in the header; it is likely responsible for the header's variable length.

At any rate, this is easy to add as a string section using Bowcaster. This is the last check in `abCheckBoardID()`.

The Mystery Header So Far

Here's a diagram of what we know about the header so far.

Byte	
0-3	Magic: "*#$^"
4-7	Header Length
8-11	
12-15	
16-19	
20-23	
24-27	
28-31	
32-35	
36-39	Header Checksum
40-variable	board_id "U12H192T00_NETGEAR"

That's four fields identified, for a total of 30 bytes. 28 bytes remain. Although the `abCheckBoardID()` function only inspected these four fields, it did populate several integers in the global header_buf structure. It remains to be seen how these fields get used.

Based on this information we can enhance the Python code to add the necessary fields. Updated code in part_6 of the git repo looks similar to:

```
from bowcaster.development import OverflowBuffer
from bowcaster.development import SectionCreator

class MysteryHeader(object):
    MAGIC="*#$^"
    MAGIC_OFF=0

    HEADER_SIZE=58
    HEADER_SIZE_OFF=4

    HEADER_CHECKSUM_OFF=36

    BOARD_ID="U12H192T00_NETGEAR"
    BOARD_ID_OFF=40

    def
__init__(self,endianness,image_data,size=HEADER_SIZE,board_id=BOARD_ID,logger
=None):
        self.endianness=endianness
        self.size=size
        self.board_id=board_id


        chksum=0;
        logger.LOG_INFO("Building header without checksum.")
        header=self.__build_header(checksum=chksum,logger=logger)
        logger.LOG_INFO("Calculating header checksum.")
        chksum=self.__checksum(header)
        logger.LOG_INFO("Building header with checksum.")
        header=self.__build_header(checksum=chksum,logger=logger)
        self.header=header

    def __build_header(self,checksum=0,logger=None):

        SC=SectionCreator(self.endianness,logger=logger)
        SC.string_section(self.MAGIC_OFF,self.MAGIC,
                          description="Magic bytes for header.")
```

```
        SC.gadget_section(self.HEADER_SIZE_OFF,self.size,"Size field representing length of header.")

        SC.gadget_section(self.HEADER_CHECKSUM_OFF,checksum)
        SC.string_section(self.BOARD_ID_OFF,self.board_id,
                          description="Board ID string.")
        buf=OverflowBuffer(self.endianness,self.size,
                          overflow_sections=SC.section_list,
                          logger=logger)

    def __checksum(self,header):
        data=str(header)
        size=len(data)
        chksum=LibAcosChecksum(data,size)
        return chksum.checksum
```

In the next post I'll discuss other functions that parse portions of the header. I'll show how to identify what fields get used where. By the end of the next installment we'll be able to generate a header sufficient to get our firmware image written to flash.

----
[1] Wah wah...Buffer overflow.
[2] I'm pretty sure it's Fletcher32. I believe this because I asked Dion Blazakis, and he thinks it is, and that dude is smart. Also I found a Fletcher32 implementation on Google Code by Ange Albertini that gives the same result as mine. And that guy is also smart.
[3] The NVRAM configuration can be extracted from /dev/mtd14. This, plus libnvram-faker is covered independently of this series, in Patching, Emulating, and Debugging a Netgear Embedded Web Server