Broken, Abandoned, and Forgotten Code, Part 3

Author: Zachary Cutlip

In the previous posts, I talked about the hidden "SetFirmware" SOAP action in the Netgear R6200's UPnP daemon, and the weird timing games we have play to deal with UPnP daemon's broken networking code. I also discussed the haphazard parsing of the HTTP headers across multiple functions. I made a guess at what headers might get our SetFirmware SOAP request passed to the `sa_parseRcvCmd()` function where hopefully an encapsulated firmware image will be decoded.

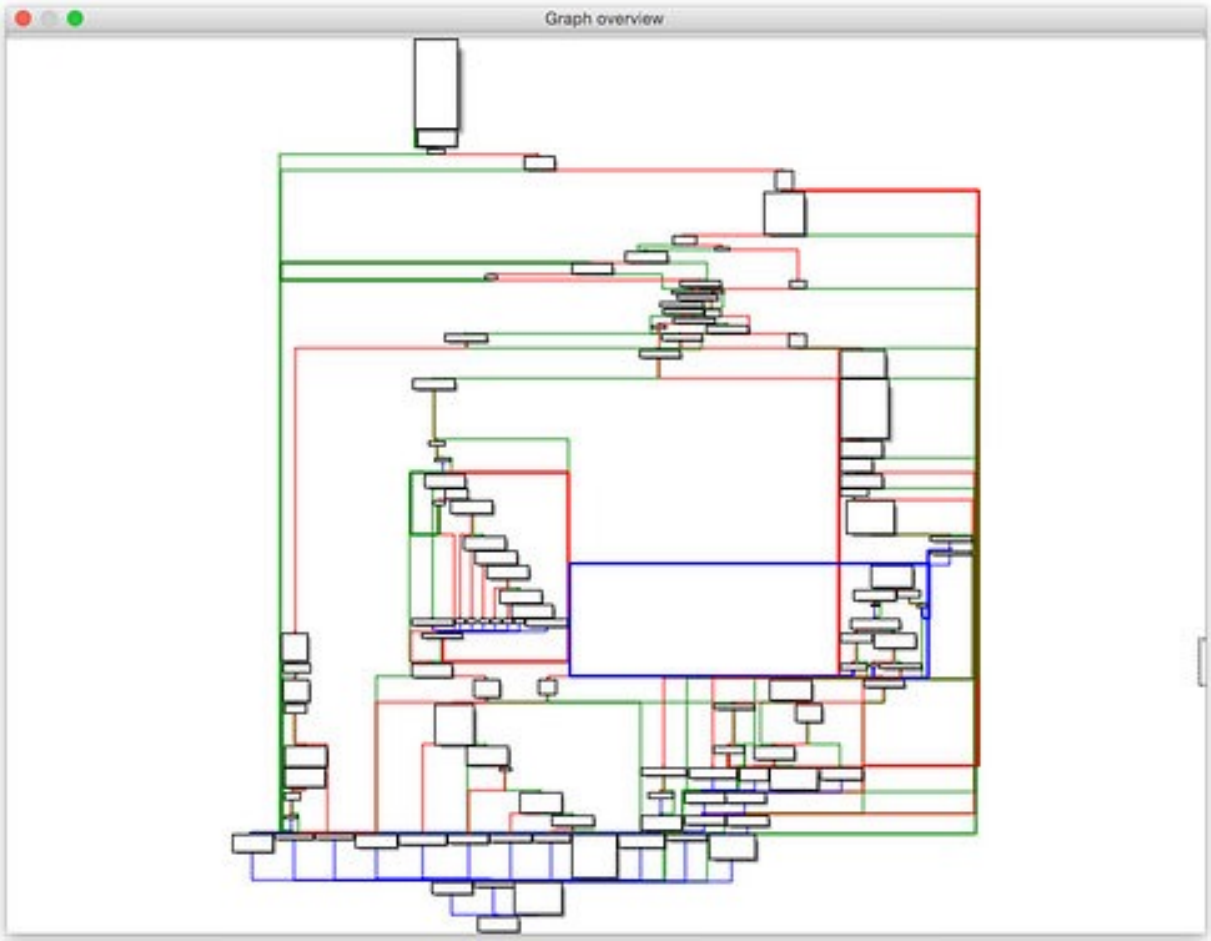
In this post I'll discuss how the `sa_parseRcvCmd()` function actually parses, or attempts to parse, the SOAP request body.

Updated Exploit Code

Previously, I published a git repository containing proof-of-concept code that demonstrates what I discussed in part 2. The repository has been updated for part 3, so if you've cloned it, now is good time to do a pull. The new code will generate the complete SetFirmware SOAP request to flash an updated firmware to the router. You can get the repo here: https://github.com/zcutlip/broken_abandoned

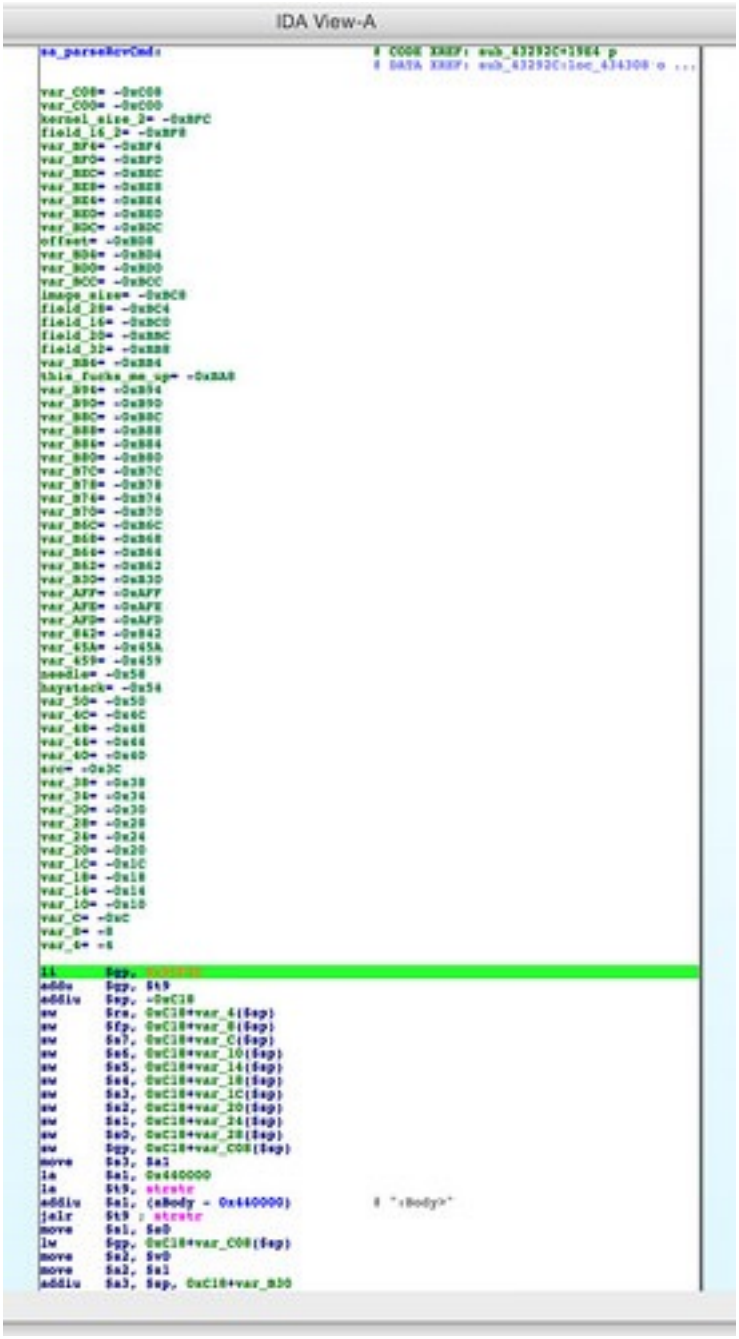
Parsing the SOAP Request Body

The `sa_parseRcvCmd()` function is large and difficult to describe. Attempting to reverse engineer the entire function would be tiresome.



Graph view of the sa_parseRcvCmd function

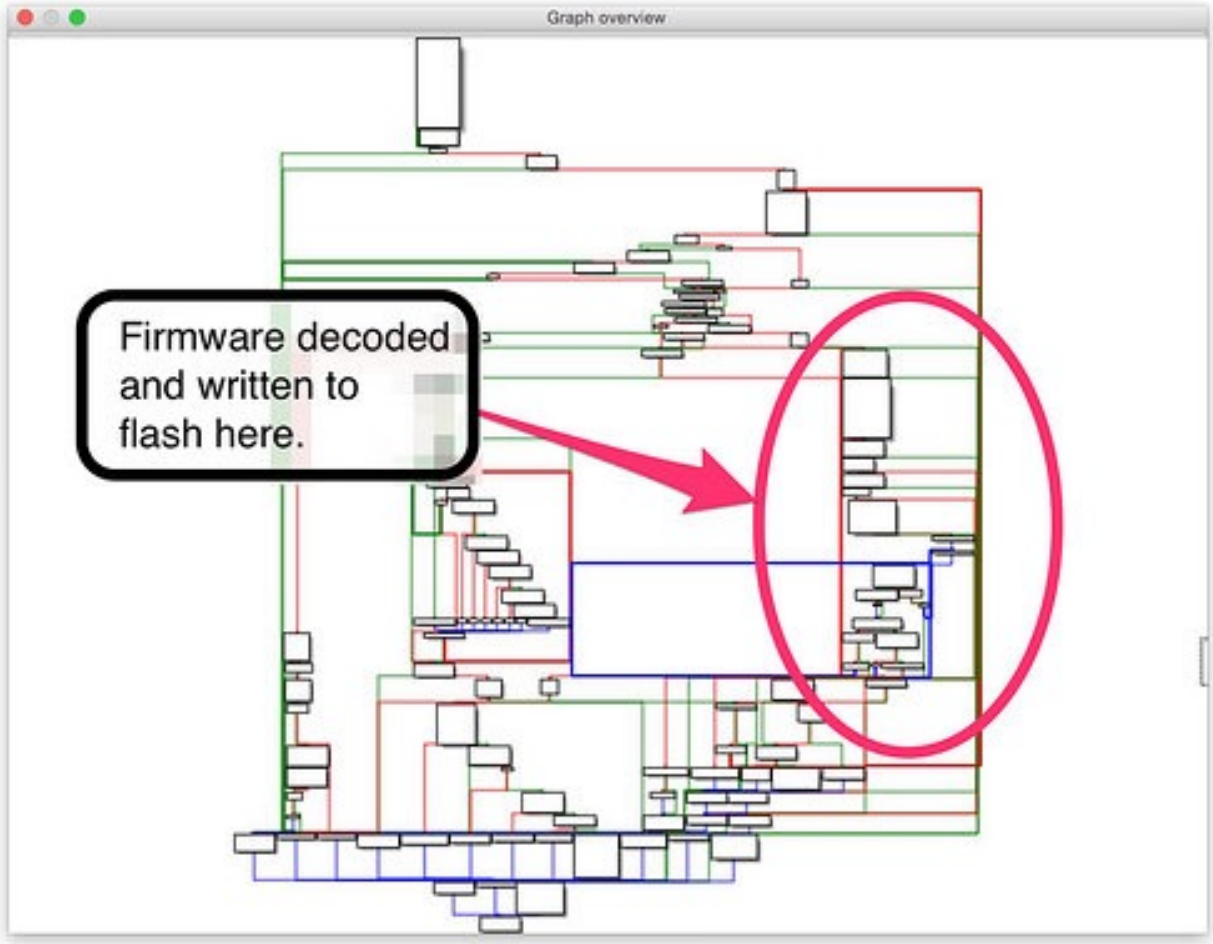
The above figure is a bird's eye view of this function. To give some perspective, the following figure is the first basic block, which includes the function prologue that sets up a long list of local variables in addition to the first bit of parsing of the SOAP request body.



Check out all those local variables.

Rather than try to understand the entire function, an easier approach is to decide where in the function we want execution to reach and work backwards from there. This way offers a better chance of finding out if the desired code is reachable, and if it is, what paths will lead there.

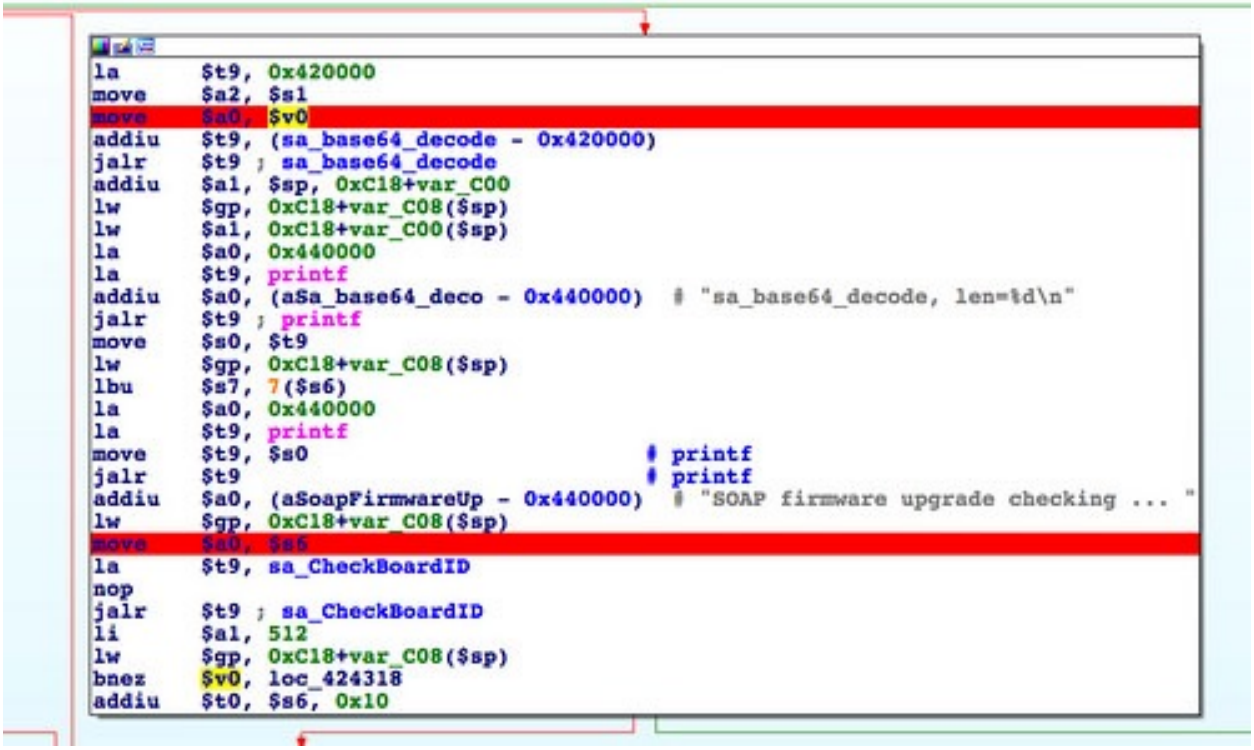
If we spend some time browsing the disassembly, we start to see what appears to be a group of blocks responsible for decoding the firmware from the SOAP request body and writing it to flash memory.



Looking even closer, we can identify the actual block where the firmware is written to flash.

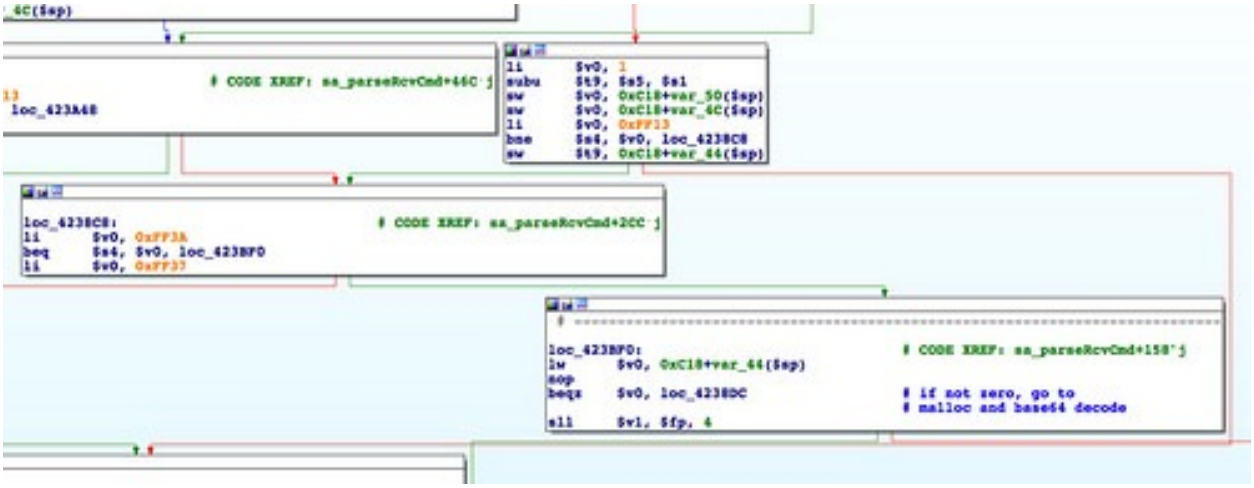
It's easy to guess that this block writes to flash memory based on the blocks that lead up to it (an earlier block opens `/dev/mtd1` for writing) as well as the error string that will be printed if the write fails. This block at 0x0042466C is our goal and the path that leads to it is how we must get there.

Working backwards, we come to a block at 0x00423C38 that appears, based on symbols and error strings, to base64 decode the firmware image.



From this we can guess that the firmware image should be base64 encoded into the SOAP request body. We might also guess that the `sa_CheckBoardID()` function in the above figure performs some sort of parsing of the decoded firmware. Once we've worked out the code path that gets to this block, we'll start working forwards again and spend some time investigating this function.

Working backwards even further, we find a cluster of blocks with many outbound paths. One of these paths (the block at 0x004238C8) leads to the base64 decoding section. This part of the function is particularly tortured, so here's the summary. This cluster appears to be a part of a large loop. On each pass through the loop, a variable is checked against a number of constants. Each comparison, if a match, results in a branch to a different path of execution. *The constant that leads to the base64 decoding operation is 0xFF3A*. While not actionable at the moment, this is worth noting.

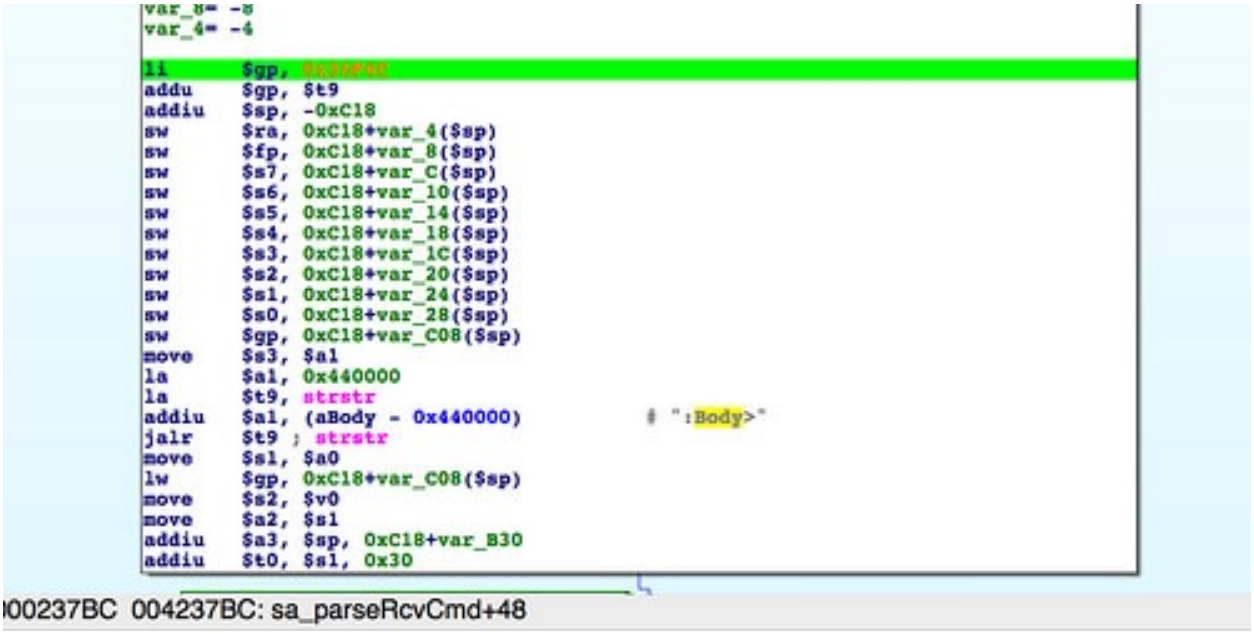


Looking for several constants. 0xFF3A leads to firmware decoding.

From there we can go backwards a little further and reach the function prologue, discussed earlier. With a general idea of the path that is

required to get the firmware decoded and written, we can start working forwards again. We now have a better idea of what code paths to focus on and what ones can be ignored.

It is at the start of `sa_parseRcvCmd()` where we find the first hints at how the actual body of the SOAP request should be structured. At the very beginning of this function, a substring search for "`:Body>`" is performed. This would find the canonical `<SOAP-ENV:Body>` XML tag that surrounds a SOAP message body. It would also find the non-canonical `<HOLY-SHIT-THIS-CODE-IS-SHITTY:Body>` XML tag. So, you know, whatever.



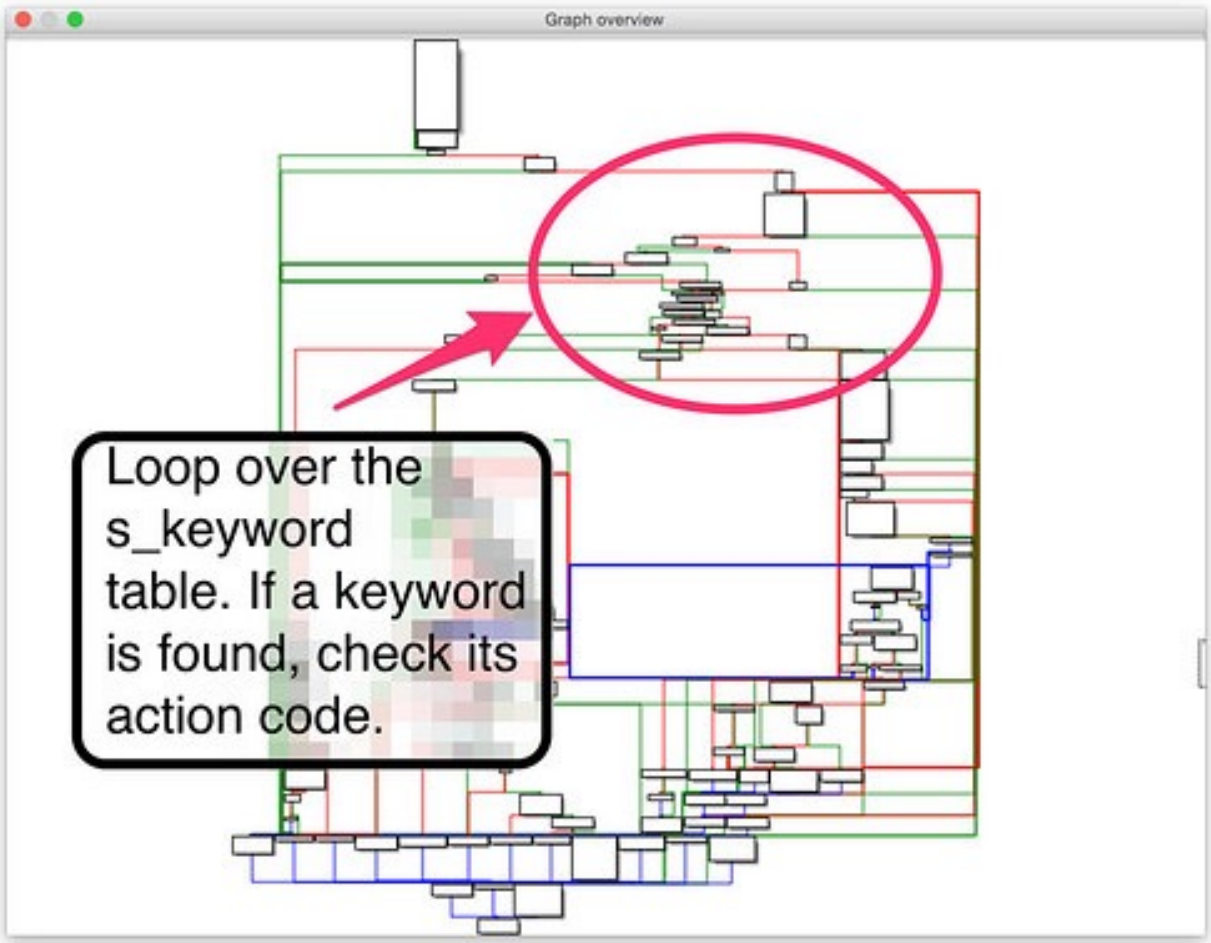
Naive string search for ":Body>".

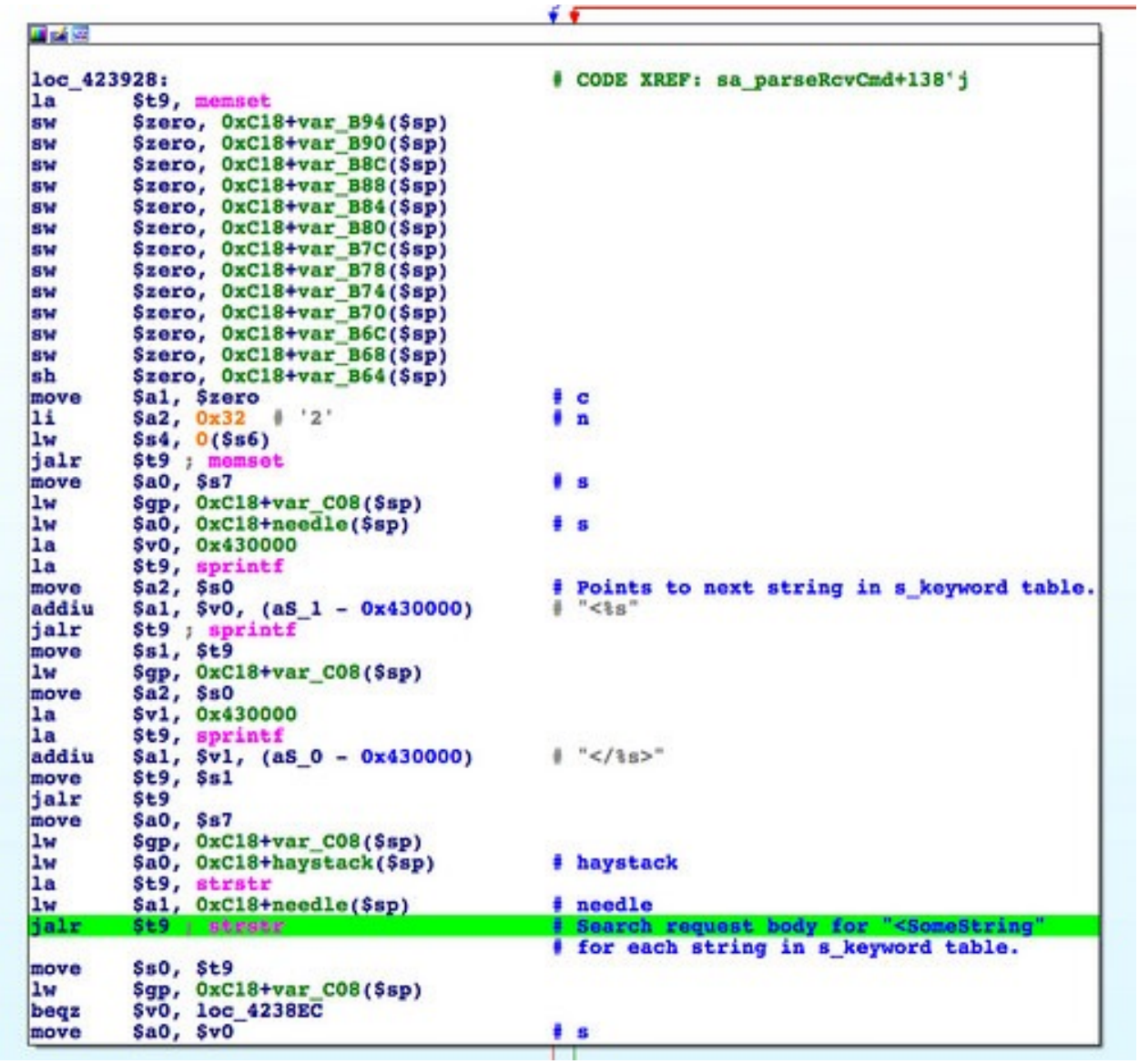
Once the body is located, the function loops over a table of strings, called `s_keyword`. This is the loop described earlier that checks for a series of constants on each iteration. The s_keyword table is an array of structs that are formed approximately like the following:

```
struct k_struct
{
```

```
    uint32_t action;
    char *keyword;
    uint32_t what_the_shit_is_this;
};
```

For each of these structures, the request body is searched for an opening and closing XML tag constructed from the corresponding keyword. If a tag is found then the keyword's corresponding action code is checked to determine the code path to take.





```
loc_423928:                             # CODE XREF: sa_parseRcvCmd+138'j
la      $t9, memset
sw      $zero, 0xC18+var_B94($sp)
sw      $zero, 0xC18+var_B90($sp)
sw      $zero, 0xC18+var_B8C($sp)
sw      $zero, 0xC18+var_B88($sp)
sw      $zero, 0xC18+var_B84($sp)
sw      $zero, 0xC18+var_B80($sp)
sw      $zero, 0xC18+var_B7C($sp)
sw      $zero, 0xC18+var_B78($sp)
sw      $zero, 0xC18+var_B74($sp)
sw      $zero, 0xC18+var_B70($sp)
sw      $zero, 0xC18+var_B6C($sp)
sw      $zero, 0xC18+var_B68($sp)
sh      $zero, 0xC18+var_B64($sp)
move    $a1, $zero                      # c
li      $a2, 0x32  # '2'               # n
lw      $s4, 0($s6)
jalr    $t9 ; memset
move    $a0, $s7                        # s
lw      $gp, 0xC18+var_C08($sp)
lw      $a0, 0xC18+needle($sp)          # s
la      $v0, 0x430000
la      $t9, sprintf
move    $a2, $s0                        # Points to next string in s_keyword table.
addiu   $a1, $v0, (aS_1 - 0x430000)     # "<%s"
jalr    $t9 ; sprintf
move    $s1, $t9
lw      $gp, 0xC18+var_C08($sp)
move    $a2, $s0
la      $v1, 0x430000
la      $t9, sprintf
addiu   $a1, $v1, (aS_0 - 0x430000)     # "</%s>"
move    $t9, $s1
jalr    $t9
move    $a0, $s7
lw      $gp, 0xC18+var_C08($sp)
lw      $a0, 0xC18+haystack($sp)        # haystack
la      $t9, strstr
lw      $a1, 0xC18+needle($sp)          # needle
jalr    $t9 ; strstr                    # Search request body for "<SomeString"
                                        # for each string in s_keyword table.
move    $s0, $t9
lw      $gp, 0xC18+var_C08($sp)
beqz    $v0, loc_4238EC
move    $a0, $v0                        # s
```

Perform a strstr() for the first string in the s_keyword table.

Inspecting the s_keyword table reveals the keyword that corresponds to the magic 0xFF3A action code: "NewFirmware".

If a <NewFirmware> tag is found inside the soap body tag, then execution proceeds to allocate memory for the decoded firmware, and then on to writing it to flash memory as discussed above.

In the previous part, I made a guess at what HTTP headers would get the request into the sa_parseRecvCmd function. At this point we now have enough information to speculate as to how the body of the SOAP request should be formed.

```
POST /soap/server_sa/SetFirmware HTTP/1.1
Accept-Encoding: identity
Content-Length: 102401
Soapaction: "urn:DeviceConfig"
Host: 127.0.0.1
User-Agent: Python-urllib/2.7
Connection: close
Content-Type: text/xml ;charset="utf-8"

<SOAP-ENV:Body>
    <NewFirmware>
        <!-- Base64-encoded firmware image goes here? -->
    </NewFirmware>
</Body>
```

If this guess is right, the function first looks for the opening Body tag. Then it looks for one of a variety of inner tags, NewFirmware being the

one we're interested in. And inside that, hopefully, it will find our base64 encoded firmware image and will decode and write it to flash. Are we almost home free? Stay tuned.