Broken, Abandoned, and Forgotten Code, Part 2

Author: Zachary Cutlip

In the part 1, I showed how the Netgear R6200's upnpd binary contains what appears to be a hidden SOAP action related to the string "`SetFirmware`". I also showed how we can get into the `upnp_receive_firmware_packets()` function if we play timing games and send our request in multiple parts.

In this part I'll describe additional timing considerations needed to avoid hanging the server. I'll also discuss sloppy parsing of the SOAP request, and I'll make some guesses as to how that request should be formed.

If you're following along, the first proof-of-concept code is available. Clone my git repo from:
https://github.com/zcutlip/broken_abandoned

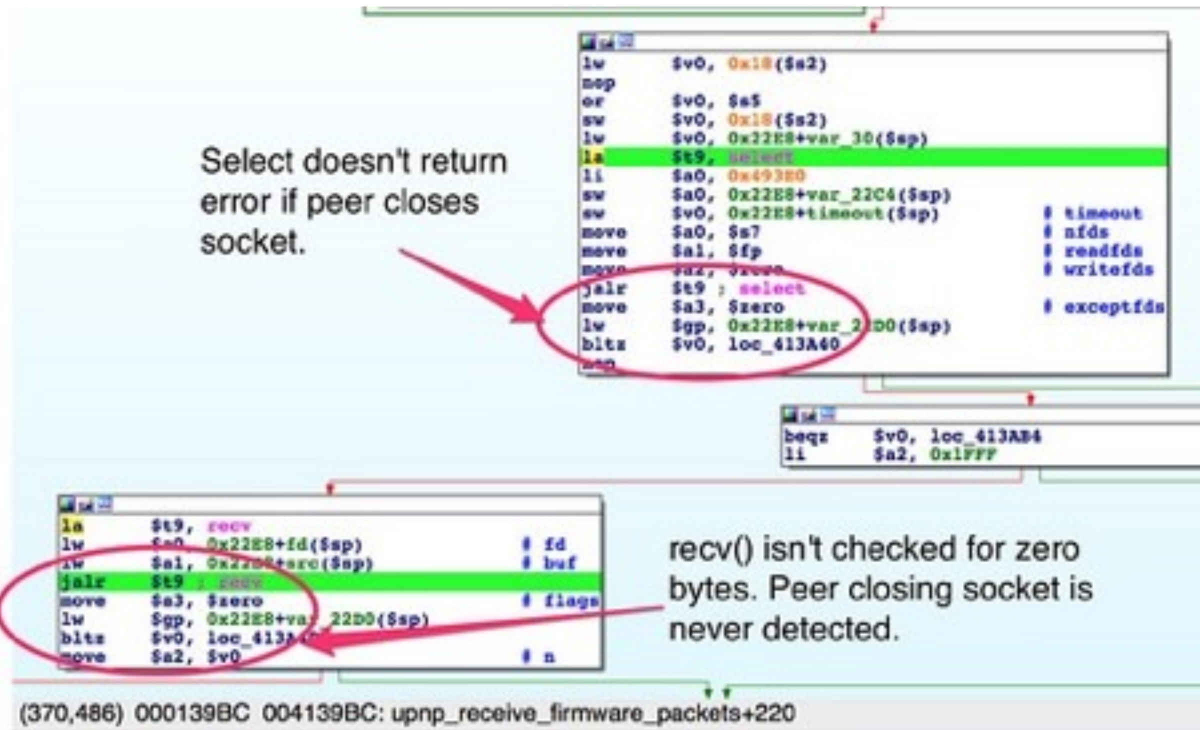
Each installment in this series that has new or updated code will have a separate directory in the repository. This week's code is under `part_2`.

Receiving Firmware Bytes

The conditions I described previously are:

- The request should be broken up into two or more parts, with the first being no larger than 8,190 bytes.
- "`Content-length:`" should be somewhere in the data, presumably in the HTTP headers (because this would make sense), but not necessarily.
- The content length should be greater than 102,401 bytes.
- The string "`SetFirmware`" should be somewhere in the data.

If those conditions are satisfied, then `upnp_receive_firmware_packets()` gets called from `upnp_main()` at 0x4144E4. In this function, a `select()`, `recv()`, and `memcpy()` loop receives the remainder of the request. This proceeds fairly sanely, with one problem.



The select() and recv() loop doesn't check for closed connections

If the client closes the connection immediately after sending the request, this function gets caught in an infinite loop. The cause for this is a little tricky to explain.

From the `select(2)` Linux man page:
A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., `read(2)`) without blocking.

If the peer has closed its end of the connection, then `select()` indicates the socket is ready because a `recv()` would not block. The way Unix TCP sockets work, when the remote end of a connection closes,

a `recv()` on that socket returns zero. In the loop, the return value from `recv()` is checked for errors (negative values), but if there are no errors, it is assumed that data was received, and the loop returns to `select()`. This results in the function looping indefinitely if the client shuts down the connection too soon.

The only two ways this loop ever terminates are (a) if `select()` or `recv()` return an error, or (b) if `select()` returns zero, indicating a timeout with no file descriptors ready for I/O. This means the requesting client must not close the connection immediately after it has sent the request. It should send the request, and then pause before closing the connection. Sleeping a few seconds should suffice.

However, there's an additional implication. Recall from before that we had to sleep 1-2 seconds in `upnp_main()` in order to get into this function. It turns out that if we slept longer, then the `select()` would time out, returning zero, and the loop would end before we had sent the rest of the request. So, while it's critical to sleep a second or two, *it's also critical to sleep no more than that*.

In review, the steps should be:

- Send 8,190 bytes or fewer, *but hold the connection open*
- Sleep 1-2 seconds, *but no more*
- Send the rest of the request, *but hold the connection open*
- Sleep a few more seconds
- Close the connection

The following code fragment sends chunks with appropriate sizes and sleep periods to get us into `upnp_receive_firmware_packets()` and to avoid getting into an infinite loop with `select()`:

```
def special_upnp_send(addr,port,data):
```

```
sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.connect((addr,port))
#only send first 8190 bytes of request
sock.send(data[:8190])
#sleep to ensure first recv()
#only gets this first chunk.
time.sleep(2)
#Hopefully in upnp_receiv_firmware_packets()
#by now, so we can send the rest.
sock.send(data[8190:])
#Sleep a bit more so server doesn't end up
#in an infinite select() loop.
#Select's timeout is set to 1 sec,
#so we need to give enough time
#for the loop to go back to select,
#and for the timeout to happen,
#returning an error.
time.sleep(10)
sock.close()
```

More Broken and Lazy Parsing

Once the entire request has been received, it is parsed, or "parsed" as it were, piecemeal, across several functions. The `upnp_receive_firmware_packets()` function calls `sub_4134A8()`. This function inspects the beginning of the received request (the first 1023 bytes, to be precise) for for the HTTP method. If the request is a POST, the `soap_method_check()` function is called at 0x413774.

```
ext:0041359C    addiu   $a1, $s3, (asc_43939C - 0x440000)  # " \t"
ext:004135A0    move    $t9, $s2
ext:004135A4    jalr    $t9
ext:004135A8    move    $s1, $v0
ext:004135AC    lw      $gp, 0x450+var_438($sp)
ext:004135B0    move    $a0, $s1
ext:004135B4    la      $a1, 0x440000
ext:004135B8    la      $t9, stricmp
ext:004135BC    addiu   $a1, (aPost - 0x440000)  # "POST"
ext:004135C0    move    $s2, $v0
ext:004135C4    jalr    $t9 ; stricmp
ext:004135C8    move    $s0, $t9
ext:004135CC    lw      $gp, 0x450+var_438($sp)
ext:004135D0    beqz    $v0, loc_413760
ext:004135D4    move    $a0, $s2
```

Checking for the POST HTTP method

```
.text:00413760
.text:00413760 loc_413760:                              # CODE XREF: sub_4134A8+
.text:00413760                 lw      $v0, 0x450+arg_C($sp)
.text:00413764                 la      $t9, soap_method_check
.text:00413768                 lw      $a3, 0x450+arg_8($sp)
.text:0041376C                 sw      $v0, 0x450+var_440($sp)
.text:00413770                 move    $a1, $s4
.text:00413774                 jalr    $t9 ; soap_method_check
.text:00413778                 move    $a2, $fp
.text:0041377C                 lw      $gp, 0x450+var_438($sp)
.text:00413780                 beqz    $v0, loc_413794
```

Calling soap_method_check()

In `soap_method_check()` several naive `stristr()` calls search for a series of strings across the entire request buffer. Based on several of the more recognizable strings, such as "`Public_UPNP_C1`", these strings are UPnP control URLs that might be requested by the POST. Although these strings may be placed *literally anywhere* (starting to sound familiar?) in the request and still trigger their respective code paths, presumably a typical request would be structured like so:

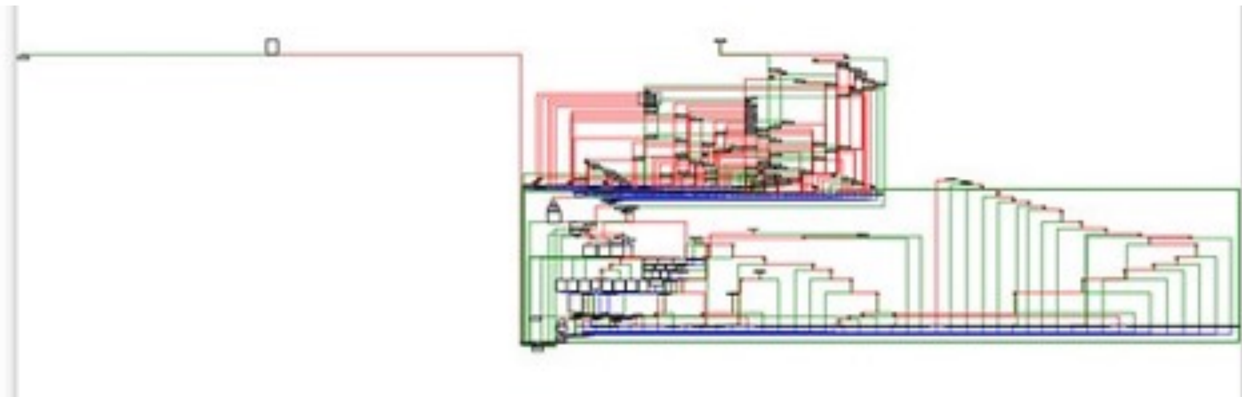
```
POST /Public_UPNP_C1 HTTP/1.1
```

One of the control URLs that is checked is "`soap/server_sa`". If that URL is found in the request, the function `sa_method_check()` is called. Note that we still don't know for certain where the UPnP daemon actually expects the "`SetFirmware`" string to be located. However, based on other, similar string references, it seems likely that this string should be part of the UPnP control URL: "`soap/server_sa/SetFirmware`".

```
.text:0041EBD4 # ---------------------------------------------------------------------------
.text:0041EBD4
.text:0041EBD4 loc_41EBD4:                              # CODE XREF: soap_method_check+170↑j
.text:0041EBD4                 la      $a1, 0x440000
.text:0041EBD8                 la      $t9, stristr
.text:0041EBDC                 addiu   $a1, (aSoapServer_sa - 0x440000)  # "soap/server_sa"
.text:0041EBE0                 move    $t9, $s0
.text:0041EBE4                 jalr    $t9
.text:0041EBE8                 move    $a0, $s1
.text:0041EBEC                 lw      $gp, 0x438+var_428($sp)
.text:0041EBF0                 beqz    $v0, loc_41EC1C
.text:0041EBF4                 move    $a0, $s1
.text:0041EBF8
.text:0041EBF8 loc_41EBF8:                              # CODE XREF: soap_method_check+1E4↑j
.text:0041EBF8                 la      $t9, sa_method_check
.text:0041EBFC                 lw      $a3, 0x438+arg_10($sp)
.text:0041EC00                 move    $a0, $s4
.text:0041EC04                 move    $a1, $s2
.text:0041EC08                 jalr    $t9 ; sa_method_check
.text:0041EC0C                 move    $a2, $s5
.text:0041EC10                 lw      $gp, 0x438+var_428($sp)
.text:0041EC14                 b       loc_41EAF0
.text:0041EC18                 nop
.text:0041EC1C # ---------------------------------------------------------------------------
```

A call to sa_method_check if "soap/server_sa" is found

The `sa_method_check()` function loops over a list of valid strings corresponding to the "`SOAPAction:`" header, and for each string in the list performs a naive `stristr()` across the entire request buffer. The string "`DeviceConfig`", if found anywhere in the request, results in a call to `sub_43292C()`. This enormous function repeatedly calls `sa_findKeyword()`, passing it the request buffer as well as various keys to be looked up in the "`s_Event`" dictionary.



The enormous graph of sub_43292c(). This function looks for keywords in the SOAP request.

The `sa_findKeyword()` function searches the request buffer for the corresponding string from the "`s_Event`" dictionary. The original

"`SetFirmware`" string is referenced by the key 49. If it is found, again, anywhere in the request, the function `sa_parseRcvCmd()` is called.

```
# ---------------------------------------------------------------------------

loc_434784:                             # CODE XREF: sub_43292C+1CF8'j
la      $t9, sa_findKeyword
nop
jalr    $t9 ; sa_findKeyword
li      $a1, 48
li      $v1, 1
lw      $gp, 0x90+var_78($sp)
bne     $v0, $v1, loc_4347AC
move    $a0, $req_buf
```

```
# ---------------------------------------------------------------------------

loc_4347AC:                             # CODE XREF: sub_43292C+1E70'j
la      $t9, sa_findKeyword
nop
jalr    $t9 ; sa_findKeyword
li      $a1, 49                         # index 49 corresponds to "SetFirmware"
                                        # in the s_Events table.
li      $v1, 1
lw      $gp, 0x90+var_78($sp)
bne     $v0, $v1, loc_434828
move    $a0, $req_buf
```

```
# ---------------------------------------------------------------------------

loc_434828:                             # CODE XREF: sub_43292C+1E98'j
la      $t9, sa_findKeyword
nop
jalr    $t9 ; sa_findKeyword
li      $a1, 50
li      $v1, 1
lw      $gp, 0x90+var_78($sp)
bne     $v0, $v1, loc_4348A0
move    $a0, $req_buf
```

Repeated calls of sa_findKeyword(). Index 49 corresponds to "SetFirmware."

The following HTTP request headers *should*, based on what we have observed so far, get the request into the `sa_parseRcvCmd()` function.

```
request="".join(["POST /soap/server_sa/SetFirmware HTTP/
1.1\r\n",
                 "Accept-Encoding: identity\r\n",
```

```
                    "Content-Length: 102401\r\n",
                    "Soapaction: \"urn:DeviceConfig\"\r\n",
                    "Host: 127.0.0.1\r\n",
                    "Connection: close\r\n",
                    "Content-Type: text/xml ;charset=
\"utf-8\"\r\n\r\n"]
```

Forming an HTTP request that would exercise the proper code path was an exercise in guesswork due to the many naive string searches littered along the way and an absence of anything resembling structured parsing.

It is in the `sa_parseRcvCmd()` function that an encoded firmware image is extracted and decoded from the request body, and assuming the right conditions are met, written to the router's flash storage, replacing the existing firmware.

Up until now, it has remained at least possible, however improbable, that the vendor may have designed a client to send the magic SOAP requests and to play the timing games necessary to exercise the firmware updating functionality. In the next part I'll start discussing `sa_parseRcvCmd()`, a complicated function with lots of code paths and lots of bugs. It is also this function where it becomes even clearer that the firmware updating capability of this UPnP server is not completely implemented and cannot actually work under normal conditions.