Broken, Abandoned, and Forgotten Code, Part 4

Author: Zachary Cutlip

In the last post, I described how `upnpd`'s `sa_parseRcvCmd()` function finds the body of a SOAP request and how it parses that SOAP request. This is a large and complicated function that processes many types of SOAP requests. I demonstrated how to work out the desired path of execution to decode and write firmware. At the end I made an educated guess as to how the SOAP request should be formed, and how the firmware should be represented in the request body.

In this post, we'll start with some prototype code that will exercise the portions of `upnpd` we have analyzed so far. It satisfies the conditions that I described in parts 1, 2, and 3. Including:

- The necessary timing games described in parts 1 and 2
- The minimum Content-Length described in Part 1
- The HTTP headers I described in Part 2
- The SOAP request body I described in part 3

PoC Exploit Code

In the previous installment, I updated the git repository with working exploit code that satisfies the above conditions. There is no update to the code for Part 4; the previous part's code is sufficient for now. You can clone the repo from:
https://github.com/zcutlip/broken_abandoned

Emulation and Debugging

Strictly speaking, you don't need to debug `upnpd` for this installment, although it may help a little. In the next several installations of this series, however, it is assumed that readers who are following along will be emulating and debugging the target processes. If you *are* following along, it's worth checking out my post on [remote debugging with QEMU and IDA Pro](). In that article, I walk you through running `upnpd` in emulation and attaching IDA Pro for debugging.

The exploit code from Part 3 allows you to specify an optional file as a command line argument to encode into the request. If you don't provide an input file, then the entire firmware data will consist of a string "A"s. This is a good starting point as a long string of "A"s is easy to identify in a debugger's memory trace.

```
MEMORY:00460D6F .byte    0
MEMORY:00460D70 aPostSoapServer:.ascii "POST /soap/server_sa/SetFirmware HTTP/1.1\r\n"
MEMORY:00460D70 .ascii "Accept-Encoding: identity\r\n"
MEMORY:00460D70 .ascii "Content-Length: 102404\r\n"
MEMORY:00460D70 .ascii "Soapaction: \"urn:DeviceConfig\"\r\n"
MEMORY:00460D70 .ascii "Host: 127.0.0.1\r\n"
MEMORY:00460D70 .ascii "User-Agent: Python-urllib/2.7\r\n"
MEMORY:00460D70 .ascii "Connection: close\r\n"
MEMORY:00460D70 .ascii "Content-Type: text/xml ;charset=\"utf-8\"\r\n"
MEMORY:00460D70 .ascii "\r\n"
MEMORY:00460D70 .ascii "<SOAP-ENV:Body><NewFirmware>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

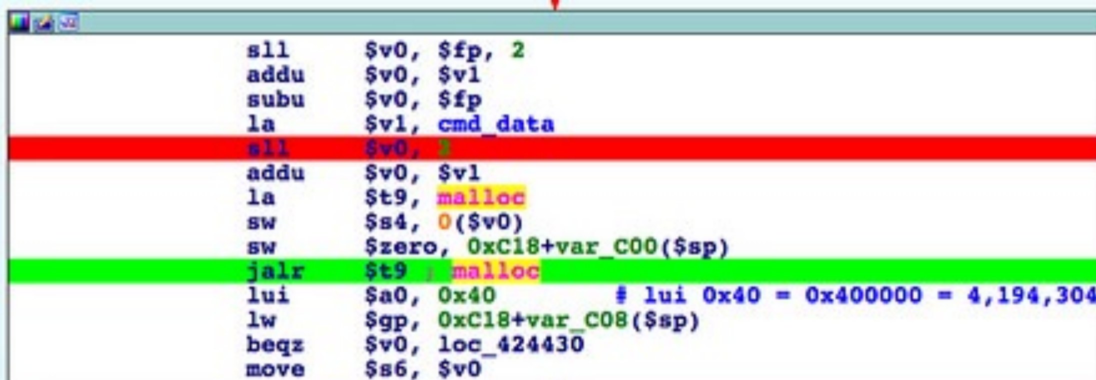
Debugger memory trace showing the SOAP request just before base64 decoding.

Crash! Hopes and Dreams Wrecked

When I got to this point in my analysis, naturally the first thing I did was encode a legitimate firmware file into the request, in hopes the firmware would be successfully written. Surprise. This was not successful. The `upnpd` deamon crashed processing the request. It was at this point in the summer of 2013 that I chucked my laptop into the river and seriously considered a career change.

Don't chuck your laptop into the river. Instead, let's figure out why the program crashes when given a legitimate firmware. I encoded a legitimate firmware file (obtained from Netgear's support website) into the SOAP request. When I sent that request to the UPnP daemon, the daemon crashed in `sa_base64_decode()`. My initial assumption was that this was a non-standard, possibly buggy, base64 decoder. I spent some time reversing the base64 decoding function. There was no obvious problem with it. Laptops were chucked.

It turns out, the problem isn't with the base64 decoder, but something more obvious. The problem is with the buffer that the firmware gets decoded into.



Allocate a 4MB buffer for decoding

In the above screenshot, we see memory being allocated. The resulting buffer is used to hold the base64 decoded firmware. Note the instruction right after the jump to `malloc()` (On MIPS, the instruction right after a jump gets executed at the same time as the jump):

```
lui     $a0, 0x40
```

For those less familiar with MIPS assembly, the `lui` instruction means "load upper immediate." This will load 0x40 into the upper half of the `$a0` register. That means `$a0` will contain 0x400000, or 4194304 in decimal. By convention, the `$a0` register contains the first argument to a function, in this case `malloc()`, resulting in a 4MB[1] buffer to decode the firmware into. The size of a typical firmware image for this device is over 8MB:

data-blogger-escaped-comment- HTML generated using hilite.me

```
zach@devaron:~/code/wifi-reversing/netgear/r6200 (0) $ ls -l R6200-V1.0.0.28_1.0.24.chk
-rw-r--r-- 1 zach zach 8851514 Jan 27  2014 R6200-V1.0.0.28_1.0.24.chk
```

In fact it's closer to 9MB. This is what crashes the program. It's unclear why the decoding isn't done in place or why the distance between the opening and closing `<NewFirmware>` tags, which is calculated right before this operation, isn't used to allocate the buffer.

```
li      $v0, 1
subu    $t9, $s5, $s1     # distance between opening and closing tag.
sw      $v0, 0xC18+var_50($sp)
sw      $v0, 0xC18+var_4C($sp)
li      $v0, 0xFF13
bne     $s4, $v0, loc_4238C8
sw      $t9, 0xC18+var_44($sp)

# CODE XREF: sa_parseRcvCmd+2CC j
# 0xFF3A corresponds to "NewFirmware"
loc_423BF0
```

In any case, this is the surest sign yet that the SetFirmware SOAP action isn't completely implemented, and likely never actually worked in production firmware. If we're going to exercise this functionality without crashing the program, it will be necessary to generate a replacement firmware image than is *dramatically* smaller[2] than the stock firmware. While possible, this is a non-trivial effort and will come with severe limitations. I'll discuss shrinking the firmware in a later post.

Before spending time on making a smaller firmware, we have a few other things to work through. We need to work out (1) what sort of validation, if any, is done on the decoded firmware, and (2) how to satisfy that validation. Further, there may be additional bugs in `upnpd` preventing a firmware from being written to flash memory. If so, there will be no point in figuring out how to shrink the firmware.

We'll start reverse engineering the firmware format in the next post.

----

[1] Technically this should be 4MiB, but in order to write that you have to say "mebibytes," which is dumb. If you hear anyone saying "mebibyte" in public, you should punch them in the face. So I'm kicking it old-school with "MB."

[2] This is the first of two potential buffer overflows that I am aware of in the firmware processing code. Some may see an opportunity here to exploit a heap-based buffer overflow. The approach I went with was to shrink the firmware to avoid crashing `upnpd`.