Broken, Abandoned, and Forgotten Code, Part 1

Author: Zachary Cutlip

Introduction

This series of posts describes how abandoned, partially implemented functionality can be exploited to gain complete, persistent control of Netgear wireless routers. I'll describe a hidden SOAP method in the UPnP stack that, at first glance, appeared to allow unauthenticated remote firmware upload to the router. After some reverse engineering, it became apparent this functionality was never fully implemented, and could never work properly given a well formed SOAP request and firmware image. If it could work at all, it would be with only the most contrived of inputs.

Someone may have thought shipping dead code was okay because an exploit scenario would be so contrived. Someone may not have considered that contrived inputs are the stock-in-trade of vulnerability researchers.

In this series, I'll describe the process of specially crafting a malicious firmware image and a SOAP request in order to route around the many artifacts of incomplete implementation in order to gain persistent control of the router. I'll discuss reverse engineering the proper firmware header format, as well as the the improper one that will work with the broken code. Together, we'll go from discovery to complete, persistent compromise.

Rules of Engagement

In order to make the challenge more interesting and to more clearly demonstrate the thesis, I decided to not take advantage of any shortcuts by exploiting vulnerabilities in the broken code path. I treated all bugs I encountered along the way as hurdles to overcome. For example, there is a buffer overflow that I will describe in a future post. I could exploit this buffer overflow to subvert the flow of execution and execute shellcode that would write my firmware, but that would be cheating. The point of this project is to show that dead code can represent a powerful attack vector, even when it is non-functional.

Target Device

The device I'll be describing in this series is the Netgear R6200 802.11ac router. Here are some specifics about the router:

- Linux based
- Little endian MIPS
- Firmware version 1.0.0.28
- Originally released in 2012
- US$200 retail price when released

I only worked with the 1.0.0.28 firmware, which I believe is the original released version. I haven't looked into later versions. That will remain an exercise for the reader. I will add that as recently as January 2015, I ordered an R6200 from Amazon, and it came with firmware 1.0.0.28 installed.

I <3 UPnP

Universal Plug and Play services on SOHO routers make for a nice attack surface for vulnerability research. UPnP services are often capable of system-level modifications that are protected only by a thin veil of obscurity. When I found strings referencing "firmware" in the Netgear R6200[1] 801.11ac router's UPnP binary I knew this daemon was going to be an interesting target. Most SOHO router exploits do not offer persistence[2], owing to their read-only storage. An unauthenticated

firmware upload is an opportunity to persist undetected on the gateway device for months or even years.

Firmware Unpacking and Strings Analysis

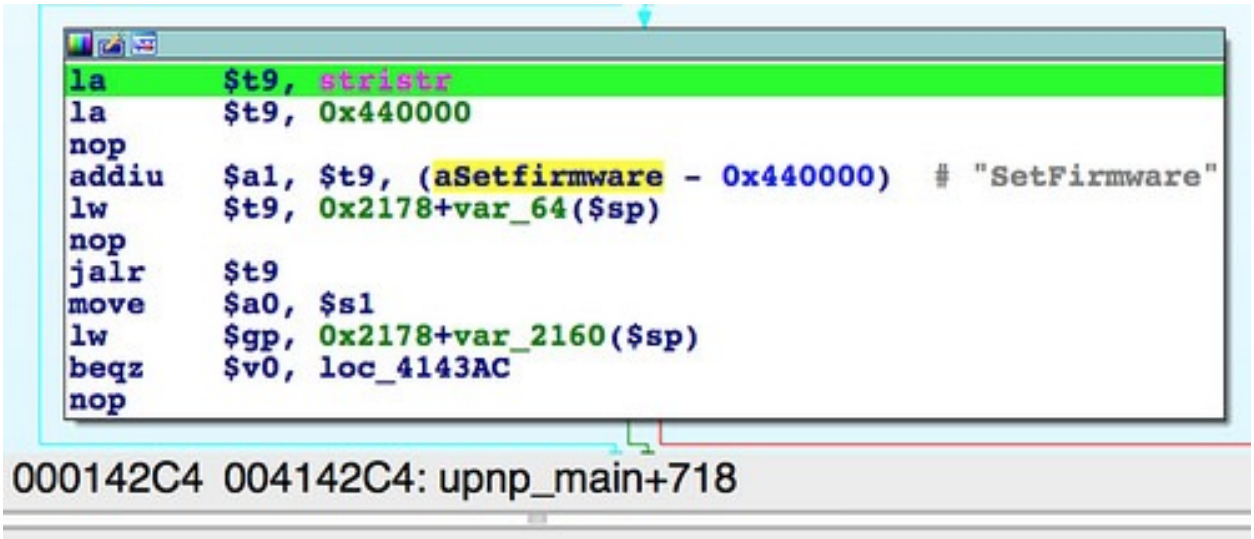
Upon unpacking the R6200's firmware, you can easily identify the UPnP daemon as `/usr/sbin/upnpd`. Source code is not available for this application, so research is an exercise in binary analysis.

Initial strings analysis of the binary reveals a "`SetFirmware`" string:

```
zach@devaron:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/reversing (0) $ strings upnpd | grep -i firmware
upnp_receive_firmware_packets
DeviceConfig_CheckNewFirmware_httpBody
/tmp/firmwareCfg
ap_firmwareVersion
SetFirmware
upnp_receive_firmware_packets
SOAP firmware upgrade checking ...
%s(%d):Board ID not matched for firmware upgrade!!
SOAP firmware upgrade pass!!
sa_setFirmware
CheckNewFirmware
UpdateNewFirmware
NewFirmware
Downloading firmware - %X bytes
        <Firmwareversion><%ap_firmwareVersion();%></Firmwareversion>
    <m:CheckNewFirmwareResponse
    </m:CheckNewFirmwareResponse>
zach@devaron:~/code/wifi-reversing/netgear/r6200/extracted-1.0.0.28/reversing (0) $
```

Strings analysis on upnpd binary, showing "SetFirmware"

Hopefully this string is somehow related to modifying the device's firmware. Static analysis reveals how the "`SetFirmware`" string is referenced in the binary:



Reference to "SetFirmware" from upnp_main()

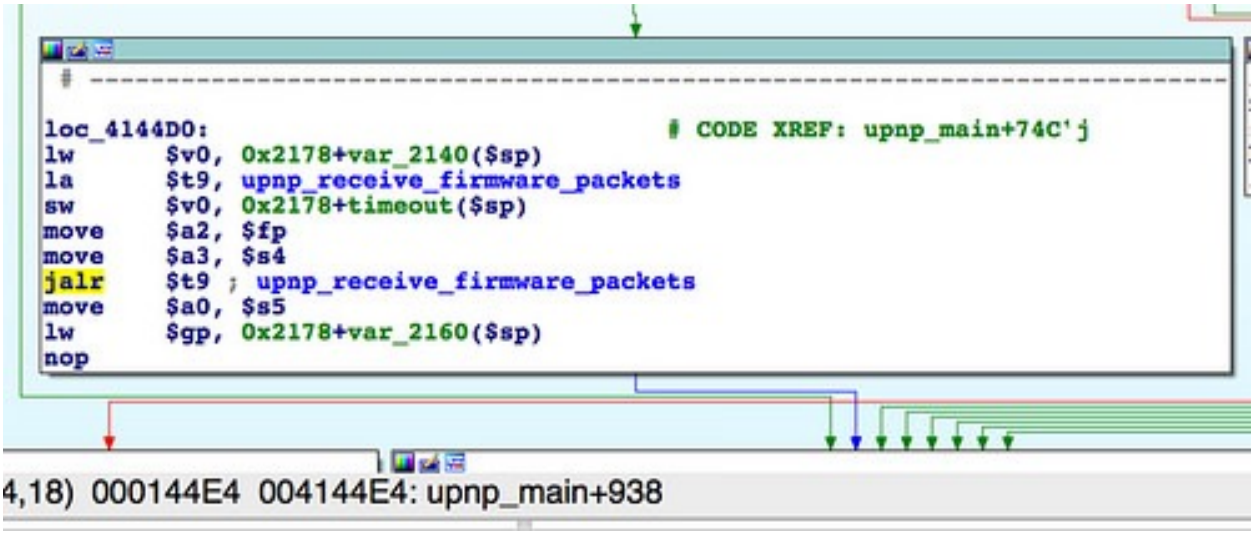
As shown in the above screenshot, "`SetFirmware`" is referenced exactly once, from `upnp_main()` at offset 0x4142C4.

Lazy Parsing

When upnpd receives a SOAP request, the `upnp_main()` function does the following:

- `recv()` from a TCP socket
- check that it received (seemingly arbitrarily) 8,190 bytes or fewer.
- perform a lazy parse of incoming requests by performing `stristr()` string searches on the received data.

The `upnp_main()` function searches for the string "`Content-length:`" literally anywhere (wtf?) in the received data. If the value following "`Content-length:`" is greater than or equal to (again, seemingly arbitrary) 102401, as checked by `atoi()` another `stristr()` is performed, searching for the "`SetFirmware`" string. *Again, this string may be anywhere in the received data*. If the string is found, `upnp_receive_firmware_packets()` is called at 0x4144E4.



A call to upnp_receive_firmware_packets()

It's worth noting the implication of these two size checks, the first for 8,190 or less and the second for a content length greater that 102,401. The request must either have a forged content-length header, or the requesting client must avoid sending the entire request in one operation. In the latter case, the request should send no more than 8,190 bytes, pause, then send the rest.

It is also worth noting that at this stage it is unclear how the "`SetFirmware`" request should be structured. It also is unclear if it should even be a SOAP request (we will proceed on the assumption that it is), or some other protocol. The only things that are known about the request 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.

This is the first bug that suggests this code doesn't actually work, at least not naturally. When you `send()` a request, you should be able to send the entire request in one operation. Your operating system's TCP/IP stack (usually in the kernel) will handle chunking the data as necessary. Further, the remote host's TCP/IP stack will handle unchunking the data as necessary. These details are abstracted from userspace code, and the receiving program should be able continue receiving until the remote end has closed the connection or until some maximum allowable size has been received. We're able to work around this anomalous behavior by sending only a small chunk of the data, then sleeping before sending the rest.

In the next part, I'll describe another bug, this time a misuse of `select()`, that *also* suggests this code never actually worked in the wild. I'll go on to describe how to make it work anyway. I'll also discuss how the broken, lazy parsing makes it difficult to know how the SOAP request should be formed such that execution follows a desirable code path.

---

[1] Although the R6200 was the primary device researched, preliminary analysis of other devices, including the R6300 v1, indicates presence of the same vulnerabilities described on this blog.
[2] It should be noted that non-persistent exploits are attractive in their own right, as the attacker may remove all traces of the compromise from the device by merely rebooting it.