Revisiting Realtek – A New Set of Critical Wi-Fi Vulnerabilities Discovered by Automated Zero-Day Analysis

By Uriya Yavniely | June 2, 2021
13 min read

On February 3rd 2021, we responsibly disclosed six critical issues in the Realtek RTL8195A Wi-Fi module, a popular Wi-Fi card found in numerous connected devices such as home and industrial appliances.

Following that successful detection and disclosure, we expanded our analysis to additional modules. This new analysis resulted in two new critical vulnerabilities discovered by scanning the modules using a unique proprietary capability of detecting potential zero-days automatically. The new vulnerabilities were fixed by Realtek, following another responsible disclosure. The vulnerabilities reside on the popular RTL8170C Wi-Fi module by Realtek affecting any embedded and IoT devices that use this Wi-Fi module to connect to Wi-Fi networks. For successful exploitation, an attacker would need to be on the same Wi-Fi network as the devices that use the RTL8710C module or know the network's PSK. Successful exploitation would lead to complete control of the Wi-Fi module and potential root access on the OS (such as Linux or Android) of the embedded device that uses this module.

To the best of our knowledge, these vulnerabilities are not being exploited in the wild. Our understanding is that the Realtek team acted promptly to patch these vulnerabilities and push the patched version to the vulnerable products.

In this blog post, we will share the technical details of the vulnerabilities, demonstrate their exploitation, and explore the automated methods we used to discover them.

The RTL8710C module

The Realtek RTL8710C module is based on an ARM Cortex M3 processor and is used for a variety of applications and by numerous devices in the following industries:

- Agriculture
- Automotive
- Energy
- Gaming
- Healthcare
- Industrial
- Security
- Smart Home

The RTL8710C as well as the RTL8195A are part of Realtek's effort to offer an alternative to the Espressif Wi-Fi modules, such as the ESP8266. Compared to the RTL8195A, the RTL8710C does not include an ADC/DAC module and has fewer GPIO ports and less computing power, while being cheaper. This means it is even more suitable for low-resource devices that are deployed in the energy and agriculture industries. For more information on this module, please refer to Realtek's documentation.

Vulnerabilities Summary

We found the module's WPA2 handshake mechanism is vulnerable to 2 stack-based buffer overflow vulnerabilities.

CVE-2020-27301 and CVE-2020-27302 require the attacker to know the network's PSK as a prerequisite for the attack and can be abused for obtaining remote code execution on WPA2 clients that use this Wi-Fi module.

Since parts of the Ameba code are shared between different Wi-Fi modules from Realtek's Ameba family, some or all of these issues may be present in other Ameba devices. For example, we found the previously-researched RTL8195A also to be vulnerable to both of the vulnerabilities mentioned above.

Vulnerabilities Technical Deep-Dive

CVE-2020-27301 – Stack overflow in WPA2 key parsing

CVSS v3.1: 8.0 (AV:A/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H)

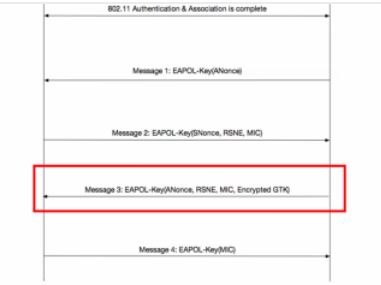
Prerequisites

This vulnerability requires knowledge of the network's PSK.

This vulnerability allows the exploitation of Wi-Fi client devices.

Technical details

As part of the WPA2 4-way handshake, a key exchange occurs at the "EAPOL" frame ('Message 3'):



In this key exchange, the Realtek WPA2 client calls the ClientEAPOLKeyRecvd to process the packet.

ClientEAPOLKeyRecvd calls the DecGTK() function, which decrypts the GTK (Group Temporal Key).

In DecGTK() an unsafe AES_UnWRAP() will be called:

```
if ( (EapolKeyMsgRecvd.Octet[20] & 7) == 1 )
{
    … // RC4 decryption, not relevant
}
v__size_1 = EapolKeyMsgRecvd.Octet[112] + (EapolKeyMsgRecvd.Octet[111] << 8);
AES_UnWRAP(EapolKeyMsgRecvd.Octet + 113, v__size_1, kek, keklen, tmp2); //
OVERFLOW
```

The above code chooses a decryption method: RC4 decryption if *EapolKeyMsgRecvd.Octet[20] & 7 == 1*, otherwise it chooses AES decryption.

EapolKeyMsgRecvd.Octet contains the 802.1x Authentication layer, so the attacker can choose the decryption method to be AES by clearing the bits given by *EapolKeyMsgRecvd.Octet[20] & 7*.

v__size_1 is also user-controlled since it is a big endian unsigned short at *&EapolKeyMsgRecvd.Octet[111]*.

If AES is used then DecGTK() calls AES_UnWRAP(), which decrypts the data in *EapolKeyMsgRecvd.Octet[113]* (The encrypted GTK bytes) to *tmp2* with *v__size_1 as the size*.

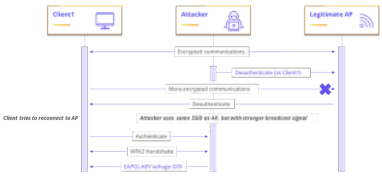
The problem is that *v__size_1* is not checked against the fixed buffer size of tmp2, so v__size_1 can be as big as 0xFFFF while tmp2's size is only 0x101. This means AES_UnWRAP can write beyond tmp2's bounds and even beyond DecGTK's stack frame and override the function's return address.

Exploitation Scenario

A real-world attacker can exploit CVE-2020-27301 on a victim client device by:

1. Sniffing Wi-Fi packets to see which wireless network the victim device is connected to and getting the SSID of that network.
2. Preparing a malicious access point (AP) that will perform the attack and has the exact SSID.
3. Sending a deauth packet to the victim device and broadcasting louder than the original network so that the device will connect to the malicious AP.
4. Perform the 4-way handshake normally, and supply an EAPOL-Key message with a very long encrypted GTK value

This is illustrated by the following diagram:



CVE-2020-27302 – Stack overflow in WPA2 key parsing

CVSS v3.1: 8.0 (AV:A/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H)

Prerequisites

This vulnerability requires knowledge of the network's PSK.

This vulnerability allows the exploitation of Wi-Fi client devices.

Technical details

There is another stack-based buffer overflow in DecGTK(), right after the vulnerable code from CVE-2020-27301:

```
if ( (EapolKeyMsgRecvd.Octet[20] & 7) == 1 )
{
    … // RC4 decryption, not relevant
}
v__size_1 = EapolKeyMsgRecvd.Octet[112] + (EapolKeyMsgRecvd.Octet[111] << 8);
AES_UnWRAP(EapolKeyMsgRecvd.Octet + 113, v__size_1, kek, keklen, tmp2);
if ( !_wrap_memcmp(tmp2, bv, 8u) )
{
    v__size = v__size_1;
    v__src = &tmp2[8];
    _wrap_memcpy(kout, v__src, v__size); // OVERFLOW

    return 1;
}
```

As we mentioned earlier, the attacker is in control of *EapolKeyMsgRecvd*.Octet and as such can force the decryption method to be AES.

v__size_1 is also user-controlled since it is a big endian unsigned short at *&EapolKeyMsgRecvd.Octet[111]*

When AES is used, then DecGTK() calls AES_UnWRAP(), which decrypts the data in &EapolKeyMsgRecvd.Octet[113] into tmp2 with v__size_1 as the size.

After the call to AES_UnWRAP(), _wrap_memcmp() is called to check if the first 8 bytes of tmp2 equals to the expected AES IV (8 bytes of 0xA6), if they do – DecGTK() will call _wrap_memcpy() in order to copy bytes from *&tmp2[8] with v__size_1* as the size, to kout which is a parameter.

DecGTK is only called from ClientEAPOLKeyRecvd(), which passes an output buffer (kout) with a fixed-size of 0x80 bytes.

The size that is passed to the _wrap_memcpy() is not checked, so setting v__size_1 to a value that is bigger than 0x80 (ex. 0xFFFF) will cause a stack overflow on the passed parameter kout.

Exploitation Demo

Since no buffer overflow mitigations are in place (ex. canaries, ASLR), exploitation is trivial.

Our test setup consists of an RTL8195A development board (the victim) and a PC connected to an ALFA network Wi-Fi adapter (the attacker). The RTL8195A is also connected to a JTAG debugger so we can get gdb output:

The PoC exploit was achieved by modifying the open-source hostapd. The attacker acts as an AP (by running our modified hostapd) and sends a malicious encrypted GTK to any client that connects to it via WPA2:

This can be seen on the right-hand-side window as "Sending malicious encrypted GTK" in the exploitation video.

The video demonstrates the stack overflow, which eventually overwrites the return address to the invalid address of 0x95f98179. This is a "random" address because the buffer goes through AES decryption, however – since the attacker has full knowledge of all the encryption parameters (the network's PSK etc.), precise control of the return address can be achieved – this is left as an exercise for the reader.

Automatically detecting CVE-2020-27301 and CVE-2020-27302

Hunting the missing function symbols

The above CVEs were detected through automated zero-day detection analysis. In this section, we would like to elaborate on this process since this specific case required interesting pre-processing, which led us to improve some aspects of our automation.

When we first analyzed the compiled Wi-Fi module binary, we did not get any substantial results, leading us to look at the binary manually. We realized many important symbols were missing because the binary code was referring to addresses that were not included in the binary:

Usually, these symbols are defined for us automatically by our emulation-based function divination engine – for example, in order to find ntohs() we can emulate a candidate function, passing a number and expecting the bitwise swapped version of that number to come out as the return value). However, in this case the actual function implementation is nowhere to be found!

After a while, we realized these function calls are calling the RTL8710 ROM, which implements a lot of important functionality, for example all "libc" functions which we often use as indicators for vulnerability sources (ex. recv) and sinks (ex. strcpy).

We contemplated guessing what these external function calls are based on their use, for example strcmp() will often be supplied with a buffer argument and a hard-coded string.

But finding symbols automatically in that manner can be prone to errors, in the strcmp() example this function might just be calculating a hash for a hard-coded string and storing it in the given buffer. Also, it requires such a call of strcmp() with a hard-coded string, one that might not be found on a different firmware.

So, we tried a different approach for finding the symbols. Fortunately, we found some sample code for this board.

Using libc functions performed by accessing a function pointer that sits in the ROM section that we're missing.

For example, to find the relevant offsets for "memcpy", we can look into –

component/soc/realtek/8710c/misc/bsp/ROM/romsym_is.so

```
SECTIONS
{
    ...
    utility_stubs = 0x8c0;
    ...
}
```

component/soc/realtek/8710c/misc/utilities/include/utility.h

After compiling the sample code, we can easily extract the offsets for each utility function:

From here, we just needed to write a preprocessor to load the function symbols based on these hardcoded addresses. Those ROM addresses do not change unless the ROM itself changes, so this technique should work well for all firmwares targeting the RTL8710 module.

Detecting the stack overflows

After all symbols were correctly defined, we ran the automated analysis again and got more substantial results, as expected. In the case of CVE-2020-27301 and CVE-2020-27302, our relevant unsafe copy scanner will find a sink function, like memcpy(), and track down a "user input" that reaches that sink function.

The platform only reports a vulnerability if it finds both a source and a sink – when looking at CVE-2020-27302, the sink, in this case, was memcpy (which was identified from the ROM mapping) and so identification for the platform was trivial. However, finding the source of a "user input" in a bare-metal firmware is much more tricky because there might not be a recv() function or other known input functions. The platform employs many different techniques and heuristics for estimating whether a particular variable comes from "user input", but in this case, the technique that proved useful is the Network Conversion heuristic.

Theoretically, any value converted by ntohs() comes from network input, and as such, the platform treats it as external data or "user input".

The problem is that both ntohs() and htons() end up performing the same code:

```
(n >> 8) | ((n << 8) & 0xff00)
```

So ntohs() and htons() might be identical to each other and in some cases be the same function.

htons() is often used when encoding data for sending it through the network, so we needed to separate between the receive (user input) and send (non-user input) flows. To distinguish between these flows, the platform will follow the data buffer backwards to make sure ntohs is working on a non-constant buffer. For example, if the platform identifies a direct constant assignment such as:

```
buf[4] = 0x1337;
...
ntohs(buf[4]);
```

The system assumes this is an htons() call (not ntohs()) and does not mark buf as user input.

Regarding the vulnerable codepath of CVE-2020-27302, we can see in the code for DecGTK() that an ntohs() operation is performed on the network data, right before passing it forward:

```
v__size_1 = EapolKeyMsgRecvd.Octet[112] + (EapolKeyMsgRecvd.Octet[111] << 8);
```

Because of this operation, the system knew to mark v__size_1 as coming from user input, which is then supplied to memcpy. Together with the system's observation that the destination for memcpy is a fixed-size stack buffer, and that no size checks are in place -> we have ourselves a stack overflow candidate.

FAQ

Q1. How do I know if my device is vulnerable?

Any version built after January 11, 2021 is completely patched against all the above issues.

The build date can usually be extracted as a simple string from the binary firmware.

For example, look for any build dates in the firmware by running the following command and observing a similar output:

```
# strings realtek_firmware.bin | grep -P '2021|2020|2019|2018|2017'
2021/05/10-17:14:47
```

Q2. Which patches can I apply to resolve the issue?

The updated versions of the ambz2 SDK can be downloaded from Realtek's website.

The latest version of ambz2 SDK (7.1d) contains patches for all the above issues.

Q3. How do I mitigate the risk if I can't update the device's firmware?

Using a strong, private WPA2 passphrase will prevent exploitation of the above issues.

Questions? Thoughts? Contact us at research@jfrog.com for any inquiries related to security vulnerabilities.

In addition to discovering and responsibly disclosing vulnerabilities as part of our day-to-day activities, the JFrog security research team works to enhance software security by empowering organizations to discover vulnerabilities through automated security analysis. For more information and updates on JFrog DevOps Platform security features – click here.

Tags: security-research, vulnerability disclosure, Security Vulnerability