Privilege Escalation for Outsiders and External Threat Actors

Security research paper by Momen Eldawakhly about *IoT Security*
Email: momeneldawakhly@gmail.com
Date: 3rd of November 3, 2022
LinkedIn

Author introduction

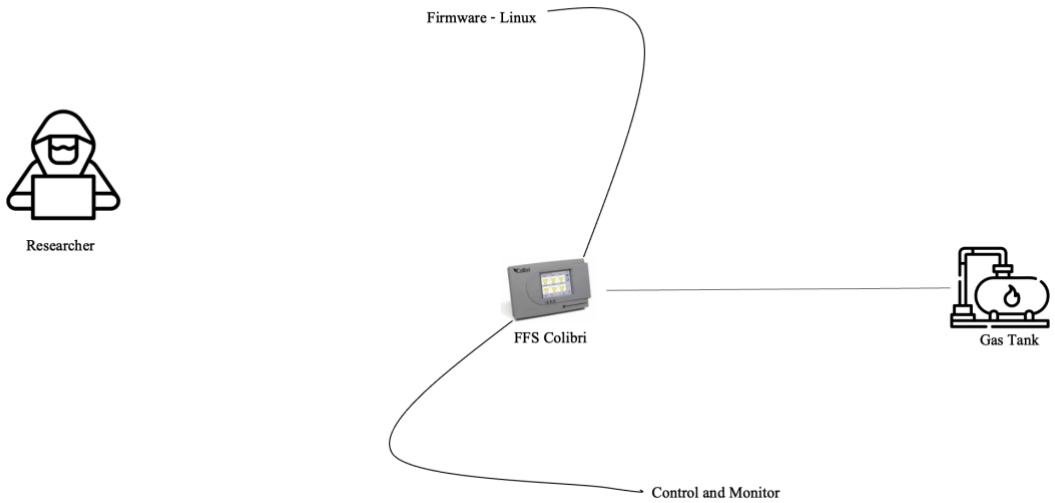
Momen Eldawakhly, Offensive Cybersecurity Officer at AiActive and Red Team Leader at Cypro AB, discovered more than 25 CVEs and participated in many security programs such as those of Google, Microsoft, AT&T, Yahoo, Oneplus, Yandex, and other programs where he discovered multiple severe vulnerabilities and was classified as the 7th researcher at the Microsoft Office Researchers 2022 Q1 Leaderboard and the 2nd researcher at the Oneplus security world rank 2021. He also secured some IoT products by finding zero-days on them, reporting these zero-days to the vendors, and helping the engineering teams fix these vulnerabilities. Products like airplane access points, fueling systems, solar power, and car management systems were in the scope of his research to secure them against security threats. He also was a speaker at many conferences, such as Black Hat, IEEE, Hacken, The Hack Summit, Wild West Hacken' Fest, and more.

When it comes to public contributions, he created the API Security Empire project that helps security testers, auditors, and developers to test and manage the attack surface of their APIs to prevent any security compromise from any external threat. He also contributed to Hacktricks and other popular references for security researchers to describe the security posture of the new technologies and the popular libraries used in the code to make the developers able to know how these libraries or technologies can affect their system or application.

Research Abstract

When we think about the privilege escalation in a system that is installed in an IoT, the first thing we start looking at is how to get into the system first, then how to escalate our privilege, but this theory most likely ends with getting access without escalating privileges because there are not that many methods that help us on that, for example, no outdated binaries, kernels, or insecure permissions that allow privilege escalation into the system, and second, after doing the analysis or the security research on the target IoT, researchers found no entrance point to that system after doing the analysis or the security research on In this case, researchers should find a new way to escalate their privileges even before entering the system. For the first time, it sounds impossible, but after this research, in which we describe a technique we made while doing security research In *FFS (Franklin Fueling System) Colibri* that allows you to do so, I'm pretty sure that you will change your mind.

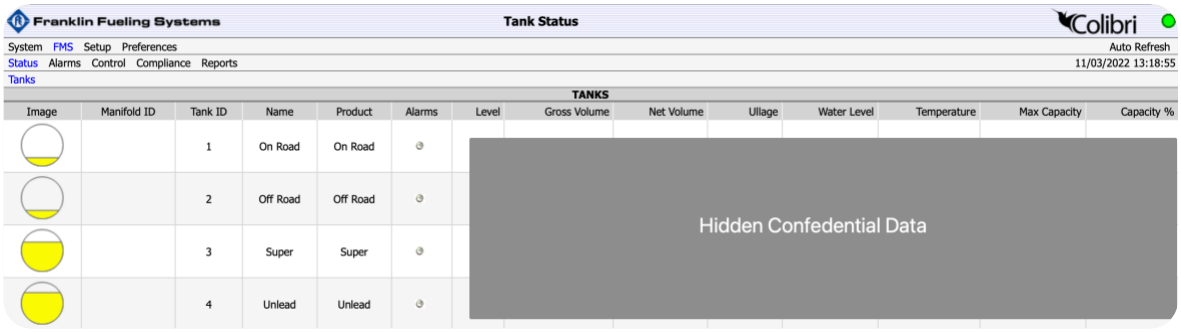
Product Abstract



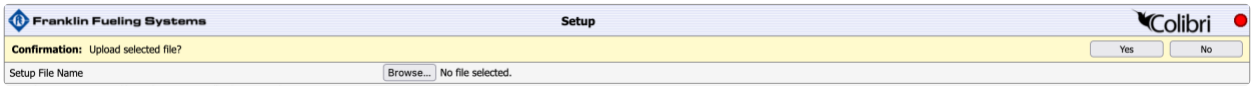
The FFS Colibri is "the ideal solution for the fueling station owner who requires basic, straightforward functionality in a fuel inventory monitoring system. The Colibri system monitors fuel density and inventory levels in up to six tanks and provides accurate, reliable information without manually taking tank readings. Additional features communicate the status of tank contents, including volume, temperature, mass, water level, and continuous tank leak detection. The Web interface feature allows authorized users access to tank information from any computer connected to the internet or wide area network, as well as custom alerts that can be sent to email or mobile devices." Gaining access to such a device allows you to have more control over it and the items it monitors and partially manages. The firmware of this device is based on Linux, so our exploitation and research will concentrate on the privileges, structure, and features of a typical Linux system.

Ecosystem discovery – the start

Modern IoTs use ecosystem interfaces to make the control of the IoT much easier and more functional. For example, some IoTs use mobile applications for door lock management, CCTV camera monitoring, or even router and access point management. While others use web applications to do the same as what we just mentioned, all these ecosystems contain a separate part of the security testing to avoid getting unauthorized access to confidential data or systems, but they can be used to attack the IoT too if an attacker is able to find a vulnerability that changes configurations or settings on an associated IoT with that ecosystem. The following is an ecosystem interface that is being used in FFS Colibri to control and monitor the tanks:



Having an ecosystem associated with IoT allows you to discover this IoT more and more, you can check the functionalities, boundaries and limitations which gives you a better and deeper vision at the IoT itself for example, what made me able to discover this new technique is an upload function we found on it allows you to upload files and firmware to the system which we will describe later how this can help:

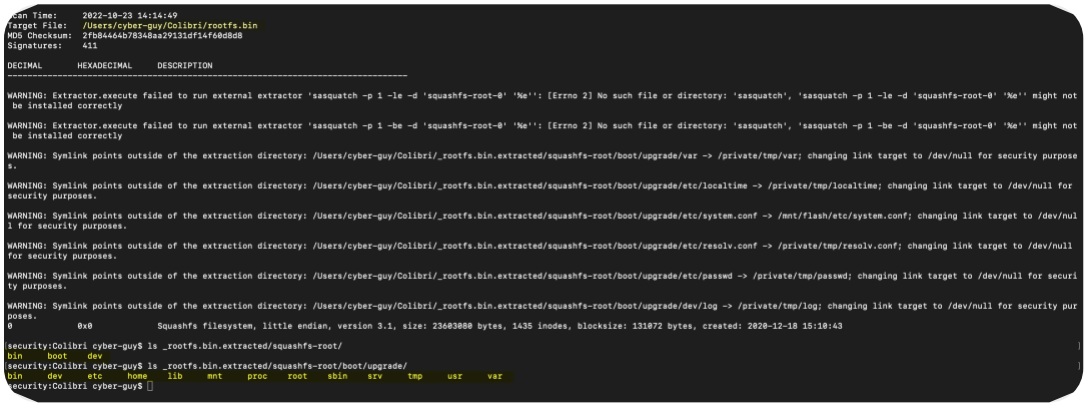


In some cases, the upload function in a web application leads to the uploading of malicious files or executables that may allow an attacker to gain initial access if he is able to find the location that the uploaded file went to, but the aim here is to escalate the privileges before entering the system. In the discovery, we always prefer to merge the backend code or binary with the ecosystem discovery to gain as much as possible vision that allows me to make an analyzing technique we called "dynamically by statically code review," where I'm doing the static code analysis but with an ecosystem in front of me for a rendered version of this firmware. The next section will discuss this deeply.

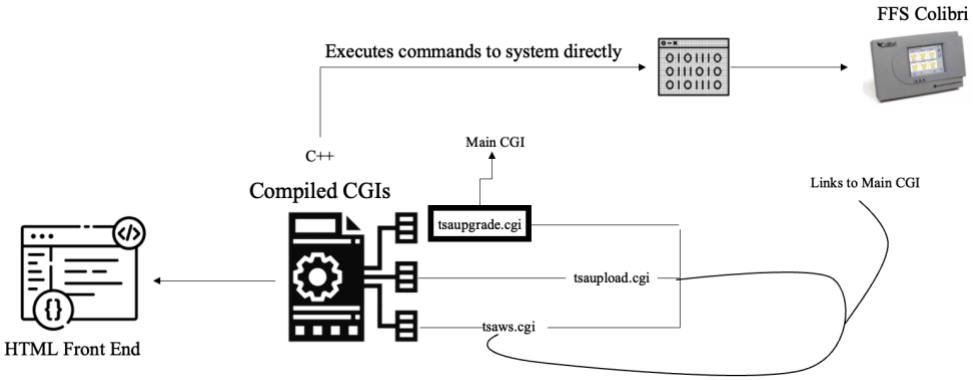
Firmware Analysis – deeper vision

Analyzing the firmware is the ideal way that allows you to understand the code workflow. Sometimes, like in our case, you can find CGI files, which are in most cases compiled, which means you will have to put in more effort to perform a binary analysis to find out how you can use them to start your attack. The firmware of this IoT was Linux-based, as we mentioned before, so our aim will be to gain access with the root privilege to take control of each part of the system *except for these processes that require kernel user access*

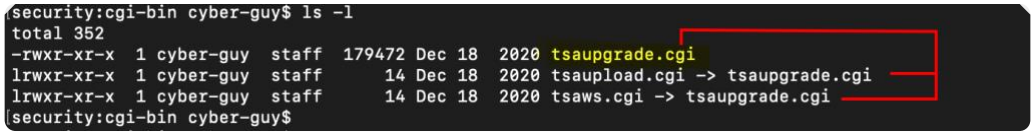
for sure. After extracting the firmware and finding the squash-fs, we started to analyze the web resources as it's our ecosystem to attack:



With further investigations, we noticed that the structure of this ecosystem can be described as follows:



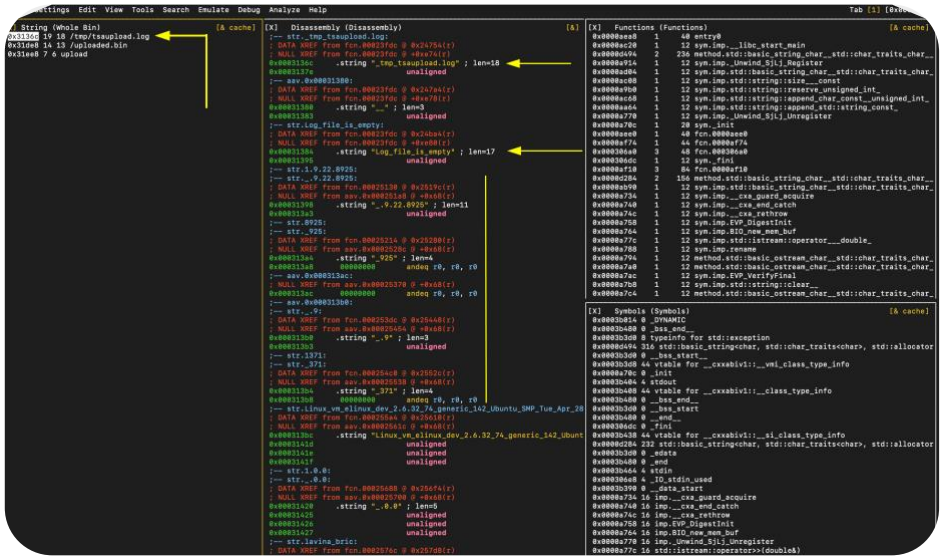
As clarified in the diagram above, the HTML there is associated with compiled CGI files, which means the main purpose of these CGI files is to command the CGI files and render the response of these CGIs into the HTML for the user. The CGI files here are compiled C++ files, as we will describe later, and these files interact directly with the system and receive commands from the web ecosystem via web commands, which we will describe later. Now, we have a deeper vision of how we can attack the system to escalate our privileges before getting into it. There are three CGI files in general, but to be accurate, there are two links to one main CGI file that is responsible for every operation in the ecosystem. You can clearly notice that as follows:



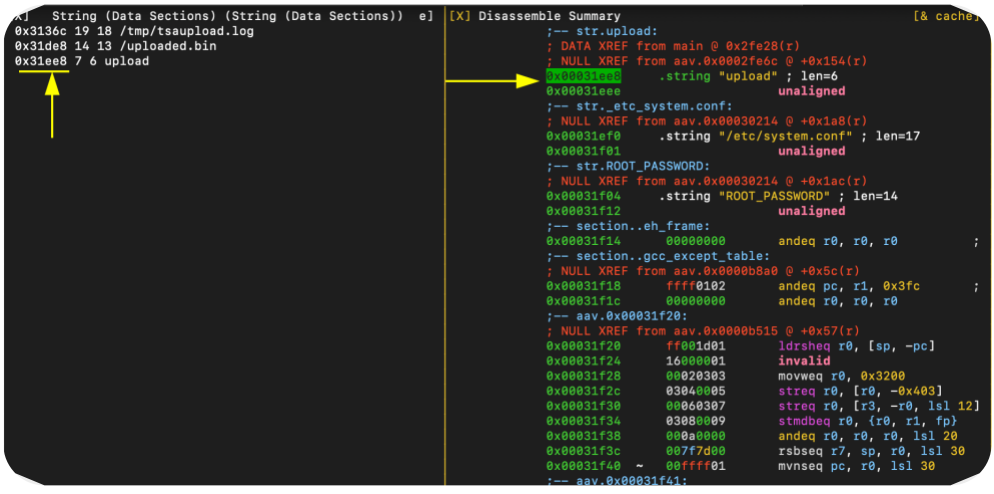
The next step is to start analyzing these binary files, but there is supposed to be only one file to be analyzed, which is the main CGI, and to do so, we will go through some steps before doing that. In the next section, we will start analyzing it to see what we can do to start our attacks.

ARM Reverse Engineering and Binary Analysis – the deepest

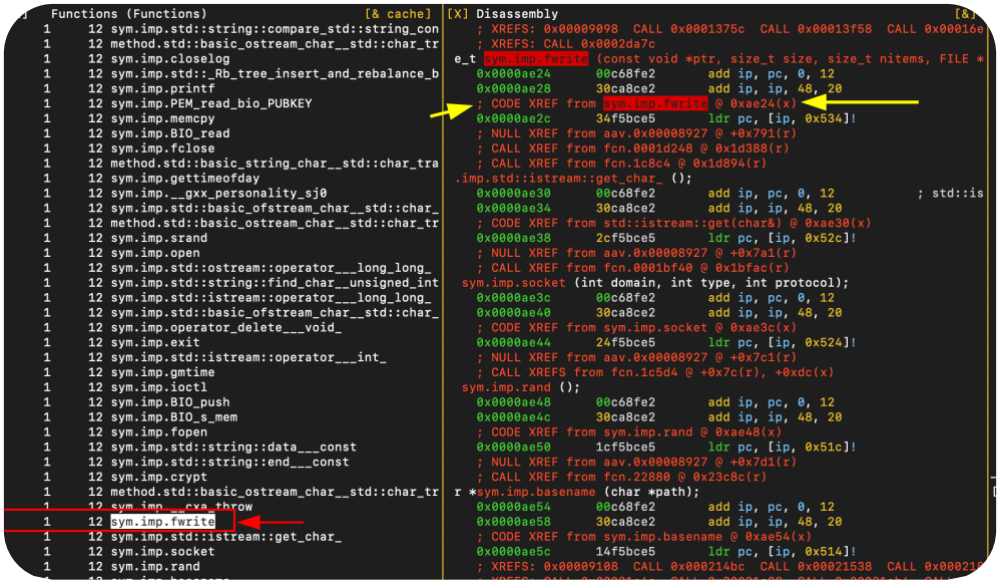
In this phase, we are going to start analyzing the binary file and, at the same time, decompile and disassemble it to start reviewing it to see what the deepest point is we can reach and to see how the CGI file handles the incoming data from the HTML. First, I'm going to use radare2 to get a quick vision of the binary of this CGI and seek in some segments about information that could be important to us, so we started to seek for the ASCII string "upload" in the whole binary file to look at the system calls and functions that are related to this ASCII and start attacking structure based on that.



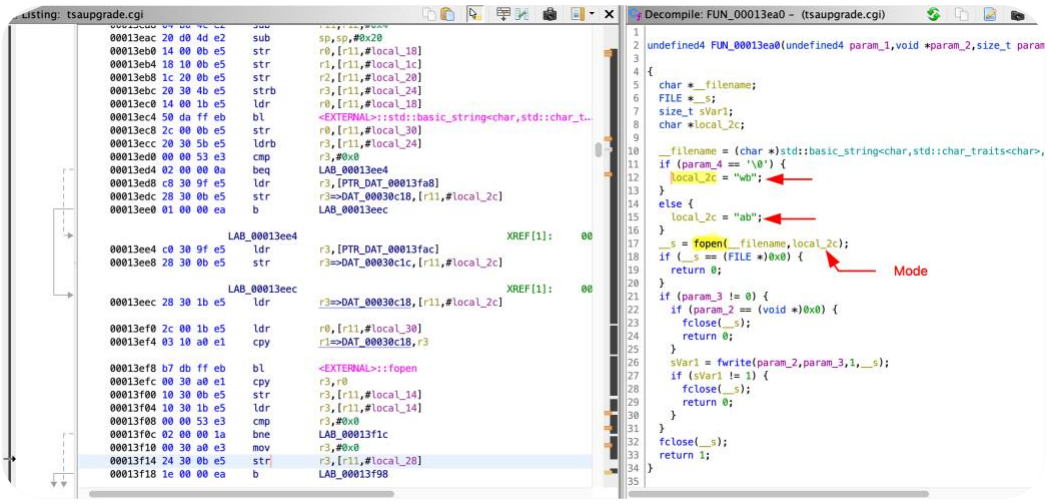
As shown, we found some results, such as "/tmp/tsaupload.log" and "uploaded.bin." Looking at these files, we can notice that the program saves the uploading logs into "/tmp/" and that "uploaded.bin" should be the destination where the uploaded file should be, so for example, if you uploaded "firmware.bin," it will be under the name "uploaded.bin" as soon as it reaches the system. On the other hand, we found a third result, which is "upload," but without a clear vision about what it is for, so viewing it, we found the following:



It's clearly visible here that the "upload" here is just a string without any indications to what exactly it's being used for, but after following the code flow, we found some relation between the "upload" ACII String and the C++ *fwrite* system function as shown:



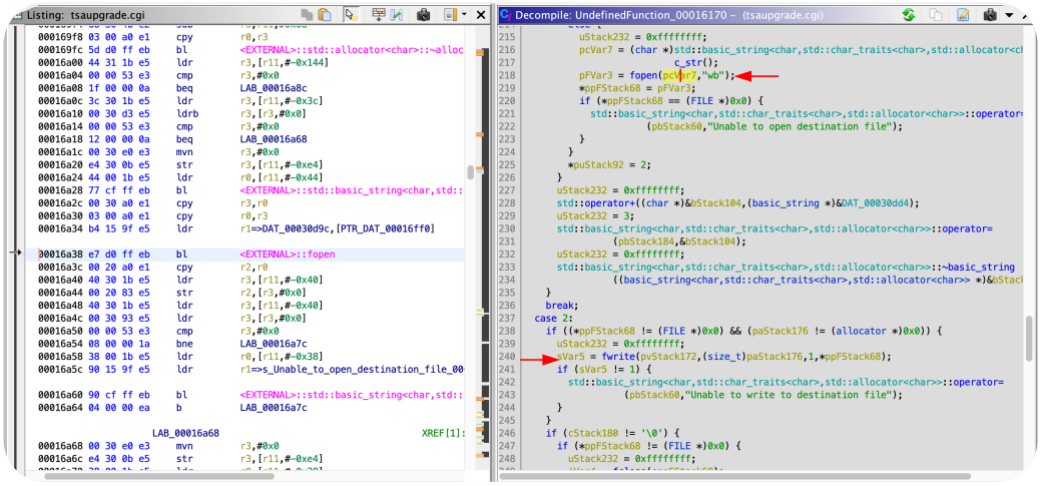
The image above describes the *fwrite* system function used in the code, and whenever an *fwrite* exists, there is an *fopen* before to prepare the modes and destination file name before writing on it. So, what we need to do now is find the *fopen* function and analyze it to see in which ecosystem function and endpoint it's being used, thus allowing us to see how we can perform our attacks.



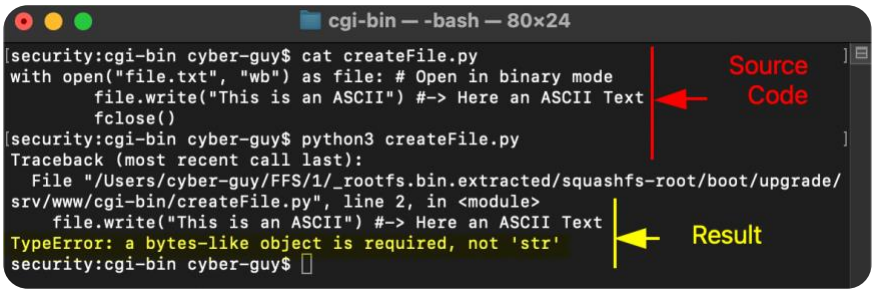
As shown below, in the program there is two modes to be included into the *fopen* function:

- wb
  - Opens an empty binary file for writing, if the file already exists, its contents are destroyed.
- ab
  - Open a binary file in append mode for writing at the end of the file and creates the file if it does not exist.

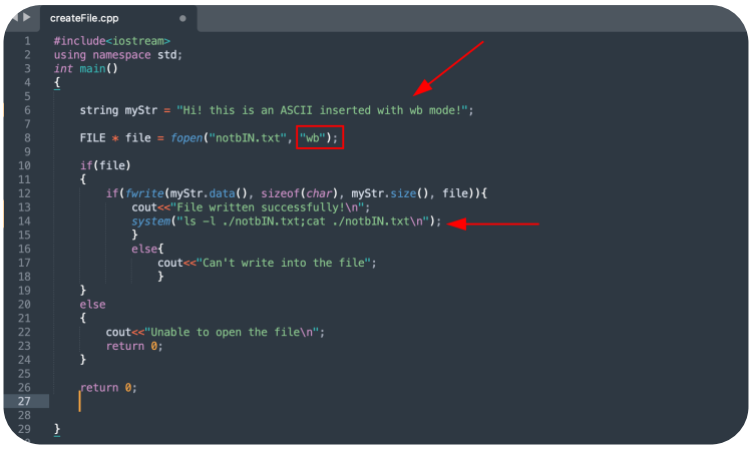
So, we need to go deep now into the code to see the place of execution and in which mode. With that, we can decide how we can exploit this behavior in the code. If we found an fopen function that uses the wb mode, writing in wb mode should not make us able to write into any files except binaries in most programming languages, so let's go further to another snippet of code that will make it clear for us.



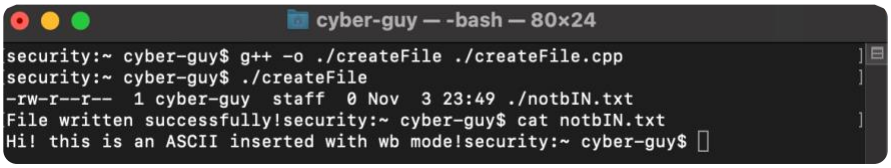
As shown above, the fopen function is called with writing into binary mode (wb), which in languages such as Python should not work correctly, as shown below:



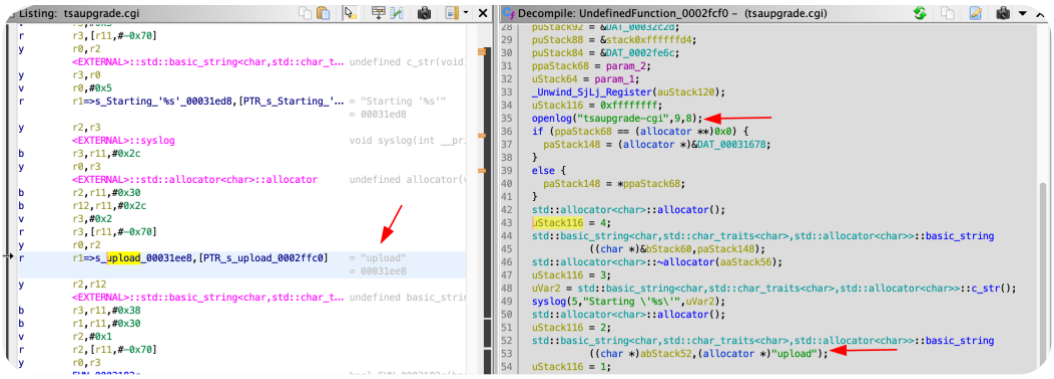
So, it's clearly understandable why this should not work in some high-level programming languages such as Python, but what about the CPP as a low-level programming language? The answer will here differ a little bit, but before we answer that question, let me create a simple program in CPP to create a file and insert an ASCII instead of binary with an fopen in wb mode to see if this will work:



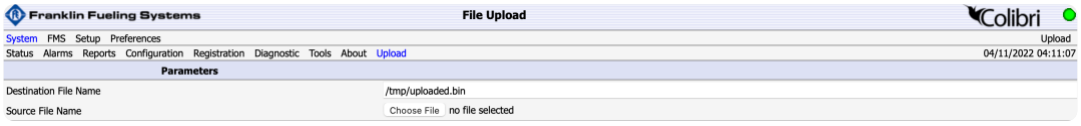
Basically, this code will create a file and insert an ASCII into it via the CPP itself, then, if this file has been created, it will list that file with a message rendered in terminal. Let us compile it now to see how it's going to work in our case:



As you can see, it has been written successfully, and the reason is stated in the fopen documentation: "The mode string can also include the letter "b" either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with C89 and has no effect; the "b" is ignored on all POSIX-conforming systems, including Linux. "Other systems may treat text files and binary files differently and adding the "b" may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-UNIX environments." So, if we are running the code on a UNIX or UNIX-like system, it should work fine without causing any errors. Now, we want to analyze that function to see in which place in the echo system it exists, and we found the answer in the following code:

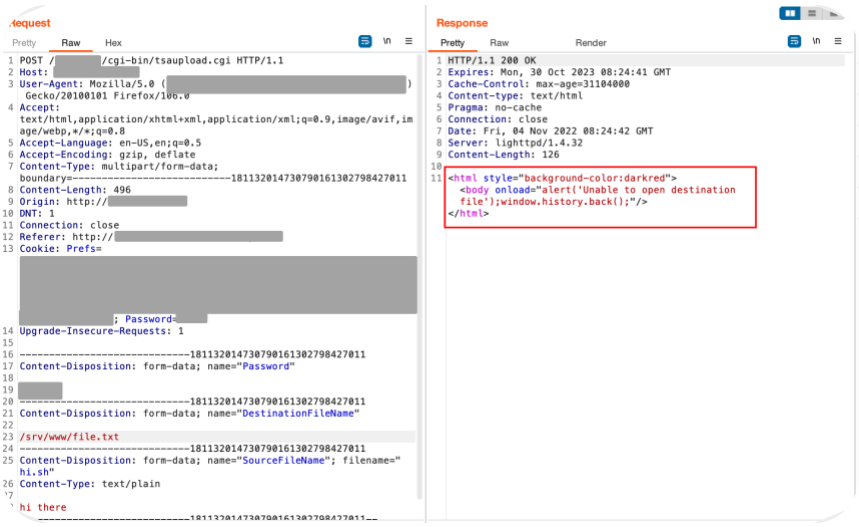


So, "upload" is the endpoint that is being used to upload the files via tsaupgrade.cgi, as we mentioned previously. Let's look at this endpoint to see what we have there:

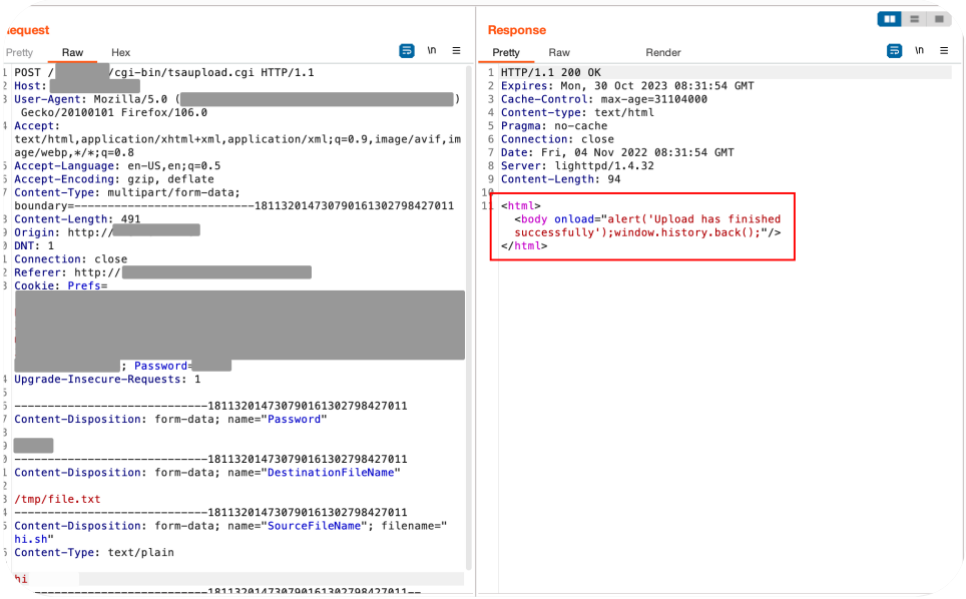


And as shown above, we can specify the destination of the file that we want to upload. And now, after examining the code, the file we want to rewrite -- let's try first to write a CGI script for example to see how the program will handle this -- first we need to know what is the full path that contains these files. We have the firmware already, so we can determine the path easily:

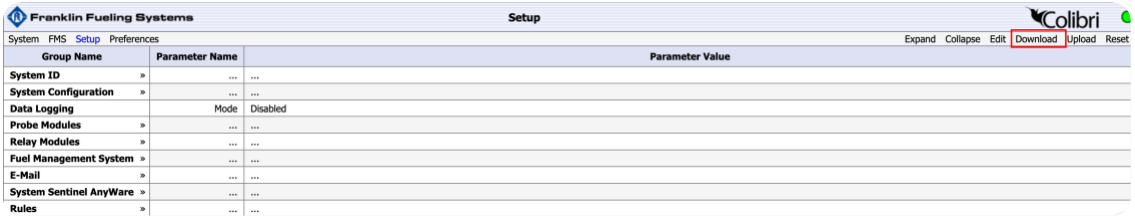
So, now when we tried to create that file in the cgi-bin, we found an error message saying "Unable to open destination file" as shown below:



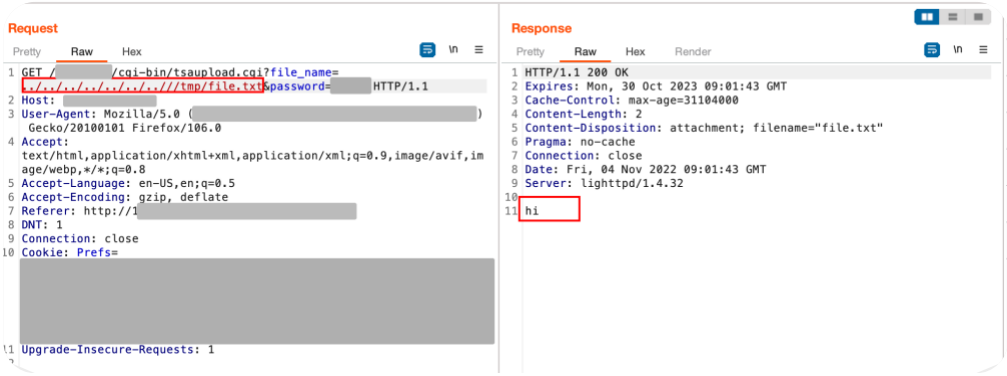
But if we upload it into the tmp directory, it will be uploaded successfully, as shown below:



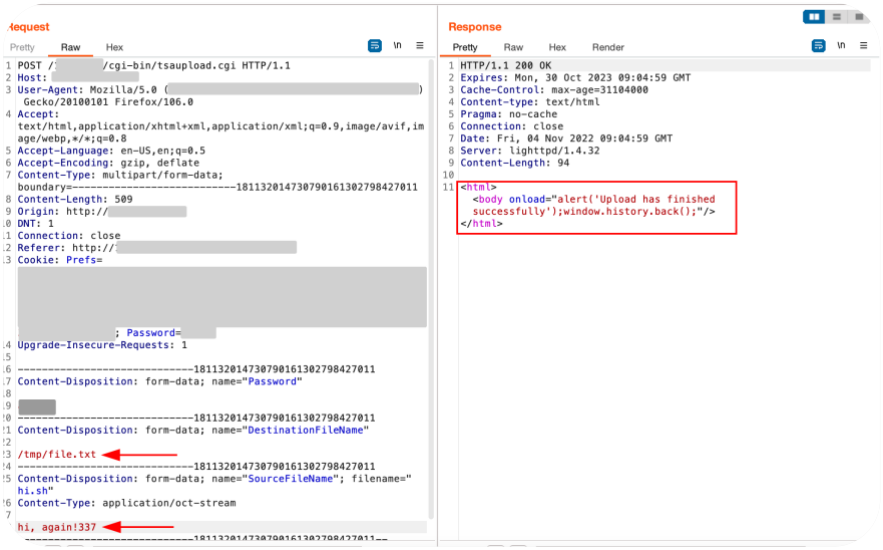
Before continuing our way, we decided to seek out any possibility of file reading from inside the system, AKA "local file disclosure." The code we analyzed according to our experience had a high possibility of having such a vulnerability. Now if we navigated to the setup, we would find a function called download as shown:



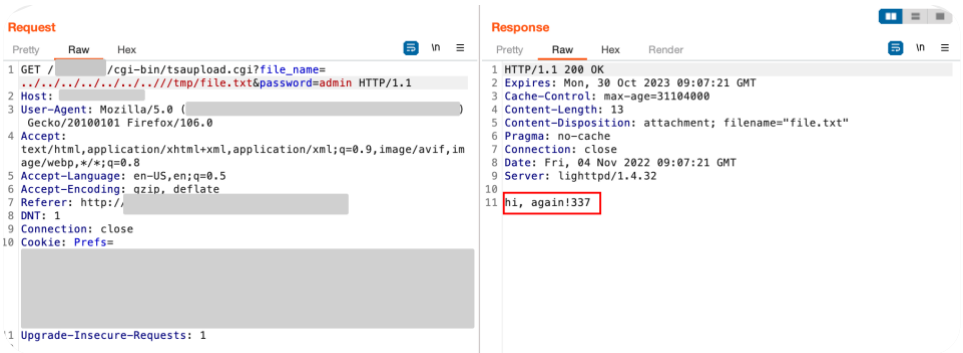
When we click that button, we can see a request is being sent to the backend to download a configuration file from the server, so we tried to attack that endpoint with path traversal LFD (local file disclosure) to get the file that we originally uploaded to the tmp before.



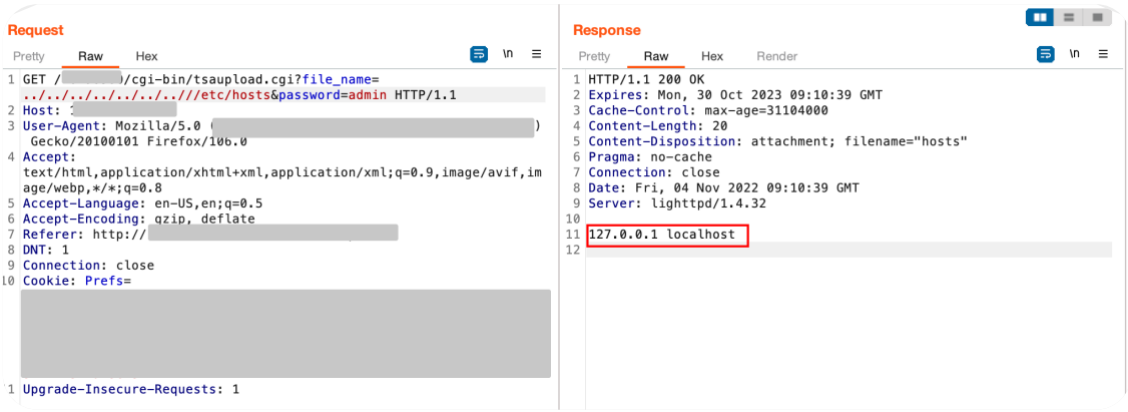
So, now we can validate our exploits by trying to create or rewrite files in the system, so let's try rewriting that file again with a different message to see if the content will be replaced or just appended at the end of the file.



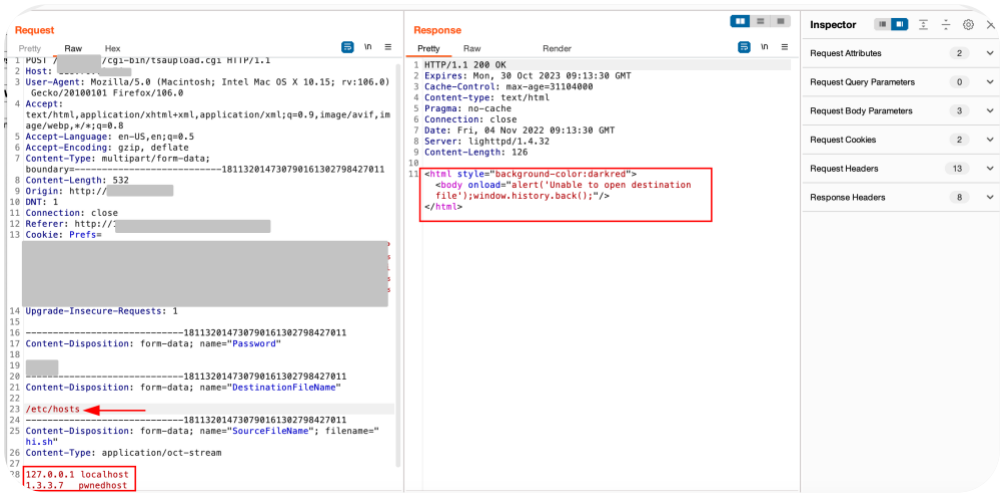
It's time now to check the result of our change, and as we expected, the file has been successfully rewritten as shown below:



Previously, we were not able to create a file in any path *except* tmp, but what about rewriting a file? Let's first get the content of an existing file from inside the etc directory, for example, the hosts file:



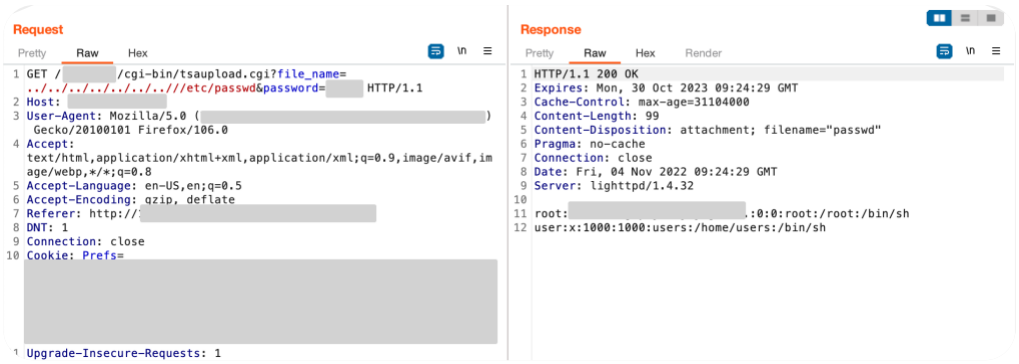
Now that we've made sure that it exists, let's now try to rewrite it with our content to see if the *CGI* will allow us to do that:



But unfortunately, we were not able to do so. When we tried to edit multiple files, we found that some files were editable and others were not! For example, the *passwd* and the *lighttpd config* files can be edited without any restrictions!

The Privilege Escalation – final thoughts

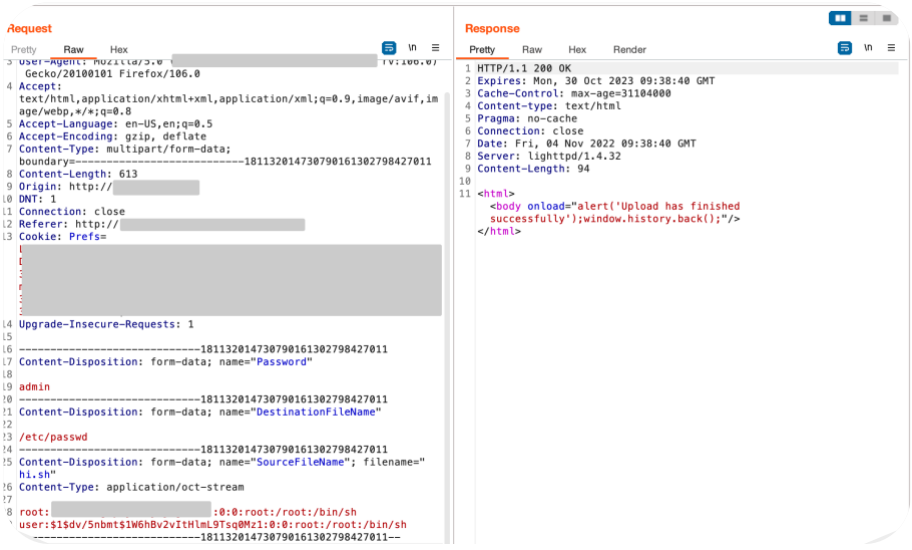
As we mentioned in the last section, on that device, there are editable files that can be edited even by low-privilege users, so first let's view the contents of the *passwd* file to check who are the available users there:



So, according to the request above, there are two users: root (high privilege) and user (low privilege). Now we will try to make the user "user" with high privilege and with our specified password. We can also change the root password, but sometimes services or logins may disable the root login for security reasons, so we will login with the other user but with the privilege of the root. First, we should generate a password that we can add to the *passwd* file. We are going to use *OpenSSL* to generate that password.

```
security:~ cyber-guy$ openssl passwd -1
Password:
Verifying - Password:
$1$dv/5nbmt$1W6hBv2vItHlmL9Tsq0Mz1
security:~ cyber-guy$
```

Now, we are going to switch the user ID, group ID, and group name for the user *user* to the typical argument values for the root user as shown:



Now, if we request the *passwd* file again, we are going to see the changes done successfully:

Final words – dropped the mic!

Finally, you may ask how to use the new user to escalate my privileges! Basically, it depends, so if you have open services such as *FTP*, then you can login with these credentials and start entering paths that require you to be a high-privilege user! If you compromised the server and found yourself as a low-privilege user in other cases where other users exist, you can switch from that user to the other user easily! There is no limitation to how you can use this to escalate your privileges. we hope you found this research paper informative as well and that it gave you more insight into how vulnerabilities can be exploited to gain more impact.

References:

https://www.ibm.com/docs/en/zvse/6.2?topic=file-fopen-mode

https://annuminas.technikum-wien.at/cgi-bin/yman2html?m=fopen&s=3

https://westechequipment.com/images/literature/Franklin_Colibri_DataSheet_Lit.pdf

https://github.com/radareorg/radare2

https://github.com/NationalSecurityAgency/ghidra