Published in Stage 2 Security

Waylon Grange
Follow
May 17, 2021 · 7 min read · Listen



# Can solar controllers be used to generate fake clean energy credits?

Last year I found myself with a little more free time on my hands but nowhere to go. I decided to use that time to look around my home for things to hack on. After surveying the IoT devices around my house, I ended up looking into the Enphase Envoy Solar Power Controller. This post outlines my findings and the 4 CVEs that came from them.

Enphase Energy provides solar energy solutions to homeowners. Their system features a cloud-based service that communicates with the Enphase Envoy solar controller and presents usage information to the users via the mobile app. As shown in the picture below, there is also a mobile installer app that is used to initialize the system.


What intrigued me most about the installer app is that it could authenticate to the Envoy controller without a user-supplied password, even after the device was installed. I took the liberty to investigate the app and found it to be a fairly simple .NET application. The app authenticates to the Envoy controller using a web API and logs in with the username “installer” it also provides a password for that account. The password is generated by calling into a native assembly library.

```
[DllImport("libemupw.so")]
public static extern int emupwGetMobilePasswd(string in_serialNumber, string in_user,
```

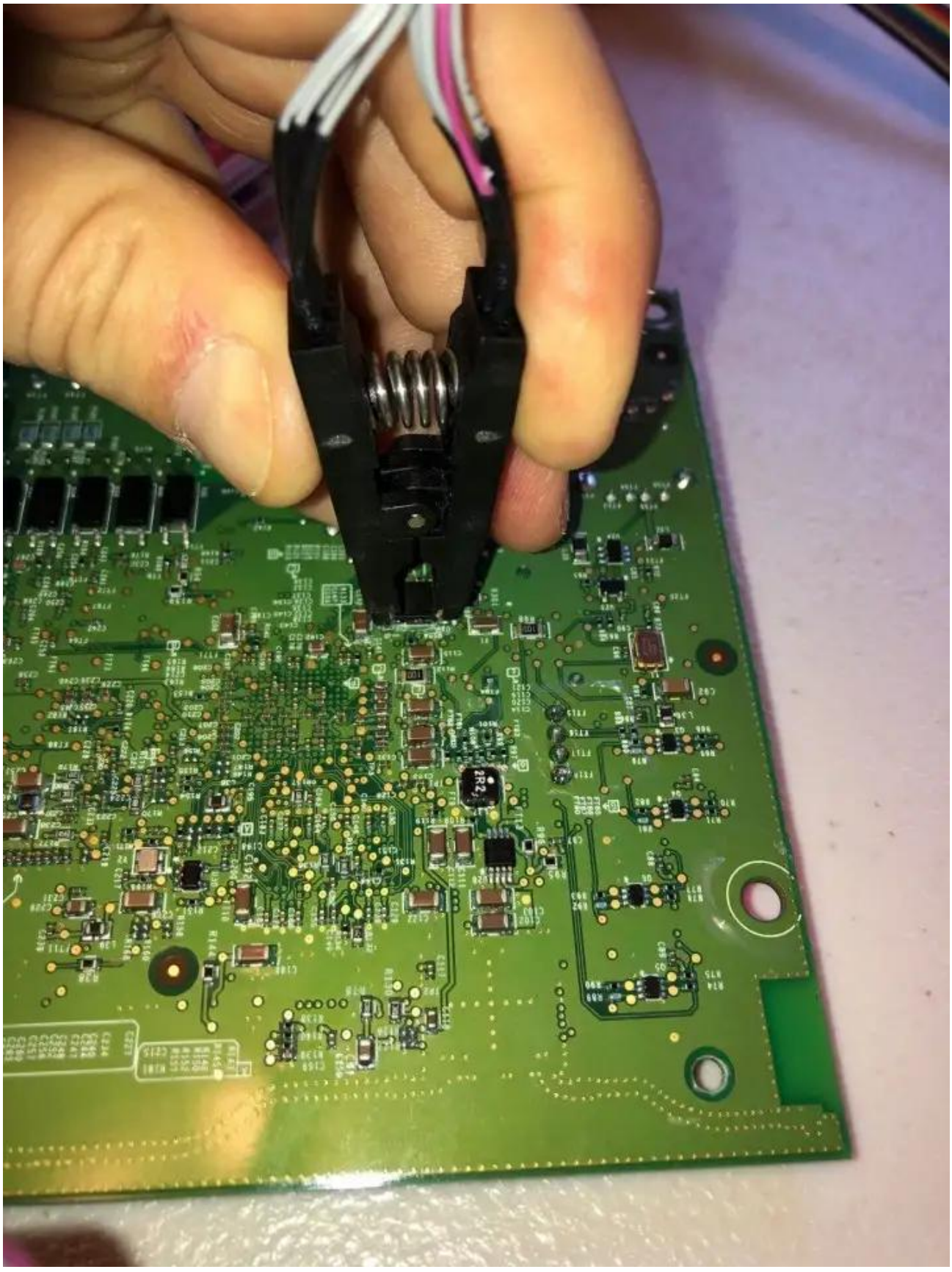
The library simply computes the MD5 hash of the string “[e]installer@enphaseenergy.com#{serial_numer}EnPhAsE eNeRgY” then uses the last 8 characters of the hex digest in reverse order for the base for the password. Using the total count of ‘1’s and ‘0’s the library then replaces all 0 and 1 characters in the password to prevent ambiguity. The simple python script below can be used to generate the password given the device serial number.

I also extract from the app the path to an AWS S3 bucket containing firmware images for the different Envoy models. It was my hope that with the image I could audit their services and authentication to gain console access to the box. However, although I had access to the firmware images they were all encrypted. Along with the images was an installer script that revealed the images are decrypted using a tool called 'eecrypt' which Catch 22, was also only available on a decrypted firmware image.

```
766     if (File.exist?("#{@sourcePath}#{file['file']}"))
767       # our file exists, now look to see if it need to be decrypted or
768       # if it can simply be moved into place.
769       #
770       if (file['file'].end_with?("eepkg"))
771         result = system("eecrypt --action decrypt --input #{@sourcePath}#{file['file']} --output #{file['localLocation']}")
772         $log.entry("#{file['file']} check1 ends with eepkg, result: #{result}")
773         # success or not, purge the original eepkg file.
774         system_sync("rm #{@sourcePath}#{file['file']}")
```

In short, I decided the only way I'd be able to get a firmware image was if I pulled it from the device itself. There are a few methods to do this, the firmware itself resides on an EMMC chip soldered to the PCB. I could attach to test pads on the board and attempt to read it there, but this only works if the data isn't encrypted. Instead, I opted to attack the U-boot bootloader.

U-boot is a second-stage bootloader that is responsible for locating and loading the Linux Kernel into memory and passing execution to it. U-boot has the capability to read and load data from ext formatted volumes and is typically where it will find and load the kernel from. The U-boot core is typically too big to be loaded on the small SoC firmware so instead resides on an external flash (typical SPI NOR flash). Locating and altering the NOR flash chip proved to be the easiest solution for me. Default U-boot settings allow a 3-second window at boot where a user could enter the U-boot console and alter boot settings. The Envoy has been modified to disabled this however, that setting can be changed by altering the SPI flash. Using a simple tool like a Buspirate or Tigard board will make this a fairly straightforward process.



Pulling the U-boot image from the SPI Flash

With U-boot console access I could read data directly off the EMMC device including the Linux kernel image and root partition. This also allowed me to grab the eecrypt binary and decrypt firmware images.

Having access to the firmware images allowed me to review their web content more easily. This led to the discovery of CVE-2020–25755. The following source code is taken from /installer/upgrade_start. This function is used to manually kick off an upgrade. The value @cm.params is a map of the URL query params and

directly controllable by the user. As can be seen on line 92 in the image below, the 'force' query parameter is passed directly to the system command with no sanitization. This simple RCE does require authentication to access but the installer credentials disclosed above can bypass that issue.

```
81
82    def start_upgrade()
83      # Make sure we are not already running an upgrade (inspired by peb/scripts/
84
85      running = `ps -axo command 2>/dev/null`.match('upgrade_start.rb')
86
87      return update_mobile_status(TaskResponse.new('1208'), false) if running
88
89      args = %w(0 mobile)
90      args << "\"#{@cm.params['force'][0]}\"" if @cm.params.has_key?('force')
91
92      status = system("upgrade_start.rb #{args.join(' ')} &")
93
94      return update_mobile_status(TaskError.new('1501', *args)) unless status
95
96      update_mobile_status(TaskResponse.new('1205'))
97    end
98
```

With command execution, my next task was to change the root password so I could SSH into the box and more easily look around. One would think this would require simply running the `passwd` command but as much as I tried it had no effect. I verified the /etc/shadow file was modified each time I changed the password but that had no bearing on SSH access nor the root user password. As I continued to trace what may be causing the issue I can upon a custom pam authentication module installed on the system called `pam_emu.so`.

As a quick recall, Pluggable Authentication Modules (PAM) are responsible for handling authentication on Linux systems. The most common of these is pam_unix which does traditional authentication using /etc/passwd and /etc/shadow but there are other common modules such as pam_krb5 for Kerberos and pam_sss using sssd. For the Enphase devices, pam_emu is set up in such a way to trump pam_unix, meaning only pam_emu can authorize user login. Thus, the contents of /etc/shadow have no effect on login access which explained my troubles earlier.

Disassembling pam_emu we find that it handles 5 different methods of authentication. Each of these it handles slightly differently. The mobile for example performs the authentication of the mobile installer password outlined above. For SSH login we're interested in the password case however http_digest is also interesting for other RCE avenues.


It turns out, the method for checking the SSH user password is dead simple. The password for these users is just the MD5 hex digest of the string “[e]{user}@enphaseenergy.com#{serial_number} EnPhAsE eNeRgY ”. Thus, armed with the user name “root” and the serial number of the device we SSH into any of these devices. Similar hardcoded passwords exist for the web authentication accounts “installer”, “pebuser”, and “enphase”. These all trace back to the pam_emu module.

I have to stop and wonder to myself why, why did they design this awful authentication hook? My speculation is that there were design constraints they wanted to meet but didn't consider the overall picture. After all, this custom authentication method does solve the problem of having a single password that works on all devices. This module does create a unique password that is based on the serial number of each device, but it comes with a large cost. Once the secret is known, that password is easily calculated and now, the password is hardcoded. Users (or system administrators) can't change that password even if they wanted to. The only way to change the password would be to remove the pam_emu module but all the remote services (including the custom mobile authentication) depend on it. It appears the designers were too hyper-focused on the one password problem to fail to see the flaws their solution introduced.

So does this just mean more pwnable IoT devices for botnets to take over? Hopefully, not. Enphase is aware of the issues and has provided an upgrade to address these issues as they should. However, there is more at stake here than just DDoS botnets. There is the threat of attackers using this to impede solar power generation but most home installations that use these controllers aren't solely powered off solar so the threat here isn't that great. Where I see the largest potential issue is with solar credits.

Enphase energy participates in the SREC program. This means users can sign up to receive solar renewable energy certificate credits which can then be traded on the private market. Enphase relies on solar energy generation data transmitted from the Enphase devices to calculate how many credits are rewarded. Thus, if an attacker can control the Enphase device they could theoretically generate bogus energy credits which could then be traded in private markets.

I didn't sign my account up for solar credits as I didn't want to get into any legal trouble but I did test the feasibility of altering data generated on the Enphase controllers. As you can see in the screenshots, my online account shows a huge increase in power! I played around with my local box a bit just to see how carried away I could get.

Hopefully, there are safeguards in place to detect when a home power system suspiciously starts reporting an unrealistic increase in power generation before it issues solar credits. If not, I'm concerned about the potential money laundering schemes to come.