Part Eleven: The Software

medium.com/@jagould2012/part-eleven-the-software-eb3590c60b6b

--

(side note, Eleven is dead. I don't believe.)

Automated Configuration

With the build complete (mostly) I moved on to the software design of the system. A few key requirements:

- All software should be installed on both computers (redundant compute design). While a board is configured to run a certain set of software on boot, it should be able to take over the role of the other board (run Pi 2 boot script on Pi 1 for example)
- It should be easy to keep the software installs in sync across boards.
- Software should be easy to test on my laptop without necessarily having the case powered up in front of me.

For this task, I turned to my friend (or arch enemy?) Ansible. Consisting of Playbooks (install scripts) and a host Inventory (list of target computers), Ansible is an automation software that remotely configures computers over SSH. It's primarily used for managing large fleets of servers, but can easily be used to maintain desktop computers also. The big advantage is that the configuration can be checked into a source repository, versioned, and updated just like a software project (referred to as "infrastructure as code").

Let's face it, if you've used Ansible, its a great tool — *once* you have it working correctly — but you always do question was the time it took to get it working worth what it buys you? However, times have changed, and my other friend / nemesis Claude knows Ansible quite well, so the upfront investment is a lot less than it used to be.

The Ansible configuration, available here, does a couple of things:

- Builds a few projects on my local machine (conveniently an ARM64 mac) to save time not building on each remote host. This includes the BBS software (Synchronet) and SDRAngel, both of which do not have pre-build ARM packages available for the Pi.
- Copies and installs all software to each Raspberry Pi and applies global settings I want on every host (disable sleep, configure Network Manager, etc).
- Applies host specific configurations — MAC address configuration, which Docker containers should start, etc.
- Configures a 3rd host, cyberdeck-testvm.local, which is a copy of Kali Linux running on my Macbook in a Parallels VM.

The test VM allows me to test new software, and develop the Ansible scripts to deploy it, without having to have the Cyberdeck powered up — and more importantly not mess up the Cyberdecks configuration. Each time I sit down to add more to the stack, I can snapshot the VM and reset it if I'm not satisfied with how things turn out. Once I'm happy with my changes, I do a pull request on Github, then run the Ansible playbooks on the Raspberry Pi boards.

At any time, with a swap of the USB cable on the front panel and running a script, Pi 1 can become Pi 2, Pi 2 can become Pi 1, giving me a true redundant system. You wouldn't want to grab the Pelican case in an emergency, natural disaster, only to find your comms down because of a $100 Raspberry Pi or $50 power supply?

Authentication

This is where the rabbit hole got *very* deep…

I was thinking through the other day the issue of how I will log into these computers:

- I don't want to make them insecure and have them log in automatically on boot.
- With just touch screens available on the system, I really hate the idea of an onscreen keyboard to type in a password.
- Making the Cyberdeck useless without pulling out the keyboard breaks the whole design (even though I do have a nice keyboard conveniently in the case).

I looked into several hardware key solutions from Yubico and Thetis that use the Fido2 protocol to securely unlock a device. These were interesting options, especially the very small Thetis Nano:

- You plug in the device to a USB port, and press a button on it to initiate the authentication. Using the right PAM module on Linux, the device unlocks.
- Some offer a built-in Bluetooth option that allow you to activate the key without plugging it into USB and send the authentication code. However, this is known to be unreliable at the login screen on Linux (apparently works very well in a browser for using the key to log into websites though).
- I could leave the key in the USB port the majority of the time, and just remove it when I want to secure the case, but this means anyone can press the button on the key and access the computer.
- The Fido2 specification requires the activation to ultimately be on the hardware key, so no options exist to use a phone app, etc to prevent the key from being used if left plugged in.

Not exactly what I was looking for. The ideal scenario to me is that I could power up the case and then use some other device, like my phone or watch, to securely login the Pi computers.

> Fine. I'll do it myself.
> — Thanos

After researching the available options, I had developed a list of requirements, but found no existing solution:

- Server software running on the Pi that was capable of logging in a pre-selected user.
- Wireless interface, no plugging in USB devices
- Fast means of authenticating from a secure device I already carry daily
- Secure login, not vulnerable to capture or replay attacks (I'm not the only person with a Mayhem board).

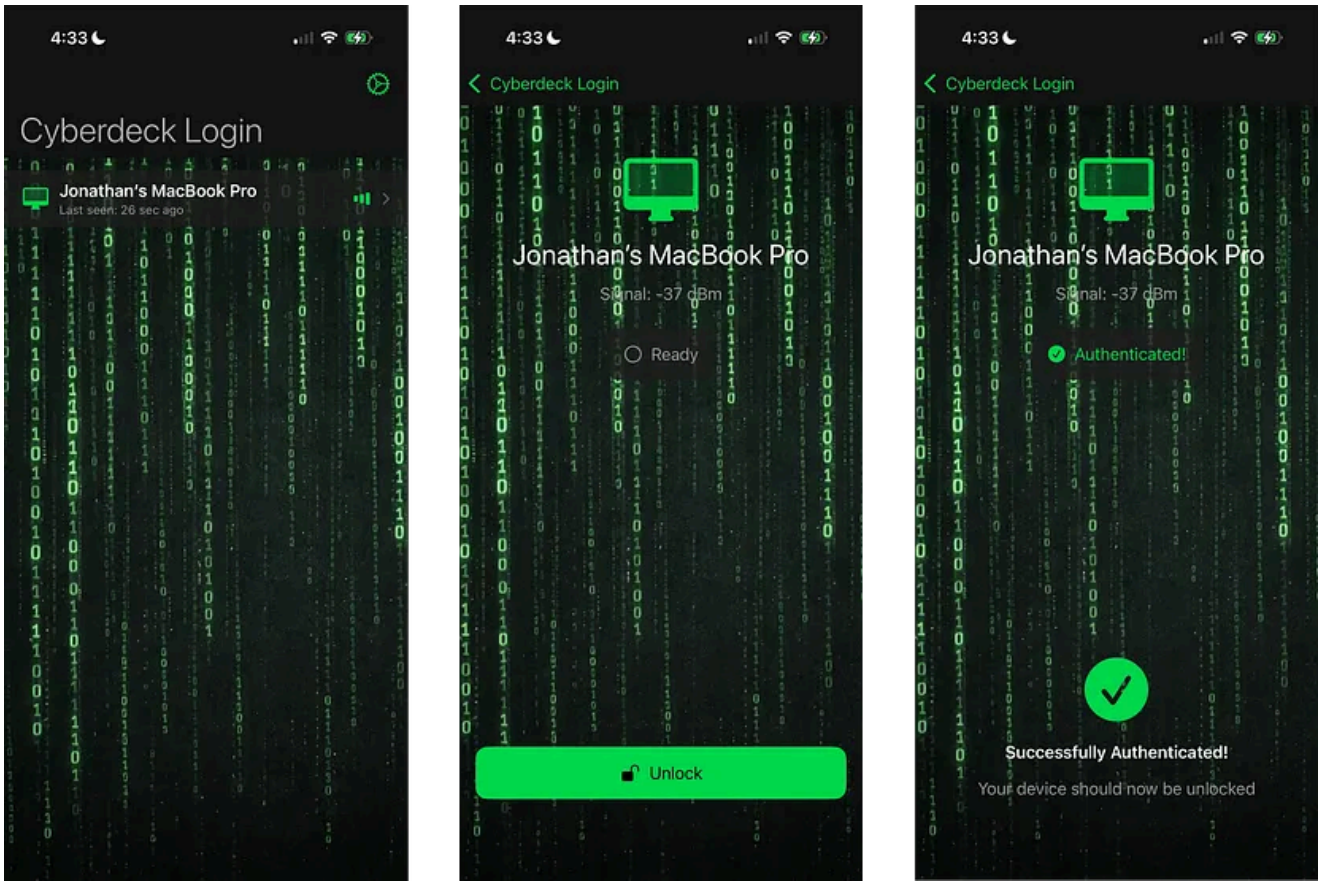
The server…

- Node.js server running in a Docker container.
- Advertises every few seconds on BLE and listens for a response.
- Generates a new nonce every 30 seconds and updates it advertisement.
- Stores a list of public keys that have been received from my client app.
- If I move a public key from the `publicKeys` directory to the `registered` directory, accepts authentication from that device.
- When a packet is received, verifies with the public key that the nonce was signed with the matching private key and unlocks the computer.
- Uses Linux `dbus` interface to communicate with the operating system.
- Immediately rotates the nonce so that the packet, if captured by a bad actor, can never be used again.

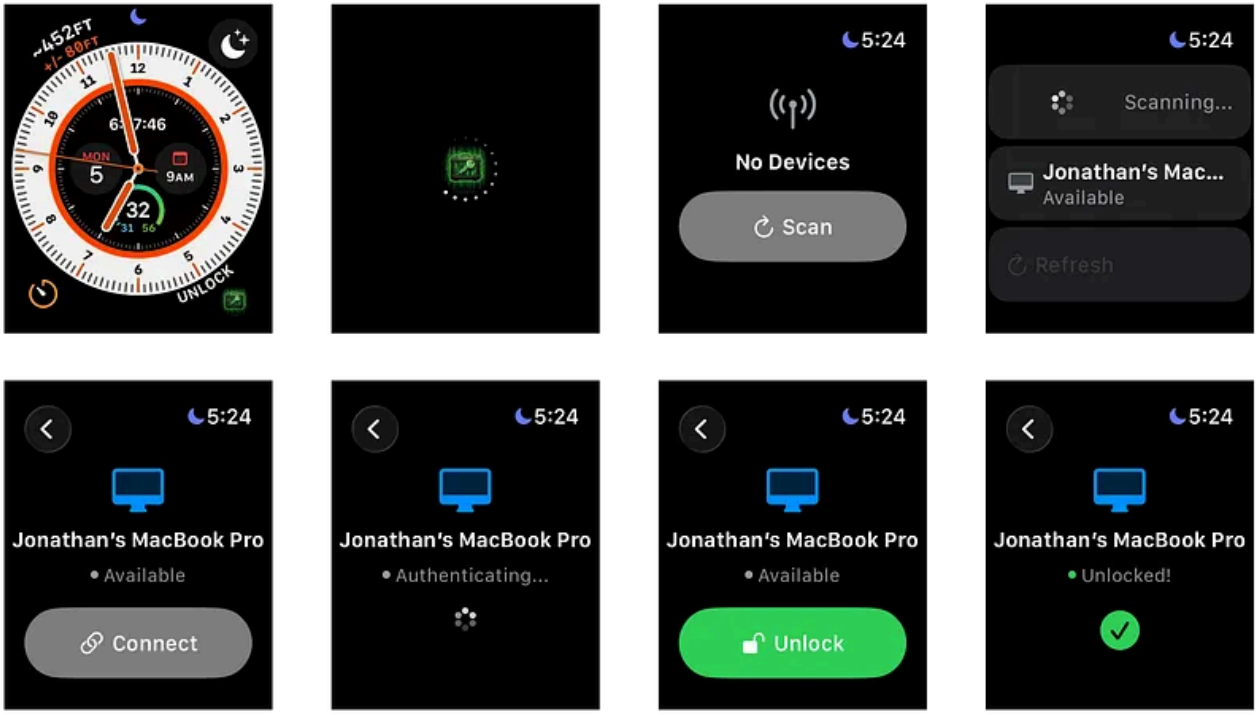
Combined with the correct custom PAM module, we now had a secure way to remotely unlock Linux.

The app…

- iPhone, iPad, and Apple Watch app capable of communicating with the server. Built with Swift.
- Generates a new public / private key on install (from the settings menu).
- When put in 'register' mode, sends the public key to the server over Bluetooth.
- Scans for nearby servers listening on BLE, allows you to connect, authenticate, and send the unlock command.
- Using Apples secure connection between the watch and phone and iCloud Keychain, share the private key with attached watches. The watch can now directly connect to the server and send its own unlock commands.
- Watch "complication" can be installed on any watch face for quickly launching the login app.



iPhone authentication app.



Apple Watch app with “complication” to launch from watch face (bottom right).

Right now I’m distributing it to my personal devices under an enterprise build, but eventually would be willing to put it on the App Store if others are interested. It could become especially interesting as a Cyberdeck companion app with additional features (reboot, power off, WiFi management).

And to be clear — I am completely aware of how overbuilt a solution this is, but it was really fun to get working. And it’s all available on the Github project of course.