NOW READING

~PABLO

Nº 04

NOVEMBER 2026

12 essays · self-hosting · 1password · adkc

PABLOMURAD.COM

CONTENTS

Nº 04
NOVEMBER 2026

01 A different kind of computer I: esoteric programming languages — 3
ESOLANGS · PROGRAMMING · HISTORY

02 Plumbing for runv.club — 12
RUNV · TOR · I2P

03 finger hi@pablomurad.com — 18
FINGER · PROTOCOLS · UNIX

04 Crypto II: My .env for everyone — 23
CRYPTOGRAPHY · SOPS · AGE

05 How I organize SSH: Tailscale for most, 1Password for the rest — 29
SSH · 1PASSWORD · TAILSCALE

06 SSH, dotfiles and secrets, part 1: the PC — 34
SSH · DOTFILES · SECURITY

07 SSH, dotfiles and secrets, part 2: a pocket vault on an Orange Pi — 44
HOMELAB · SELF-HOSTING · ENCRYPTION

08 A different kind of computer II: fantasy consoles — 52
ADKC · PICO-8 · GAMES

09 Saw That I: The Wizard of the Kremlin — 60
MOVIES · POLITICS · RUSSIA

10 Crypto III: Sending secrets through a wormhole — 62
CRYPTOGRAPHY · MAGIC-WORMHOLE · ENCRYPTION

11 Three ways to build the same wiki — 67
WIKI · PROJECTS · GO

12 Debloating my Galaxy S25 Ultra with adb, a year late — 71
ANDROID · SAMSUNG · ADB

Every essay is on the website, with its links.

A different kind of computer I: esoteric programming languages

The first part of A different kind of computer, a series about computing off the beaten path. It starts with esoteric programming languages: where they came from, Brainfuck up close with a tiny interpreter, where to begin, and how to design and publish a language of your own.



This is the first part of A different kind of computer, a series about the corners of computing that were never meant to be productive and turned out to be some of the most interesting places to spend time in. Along the way it will go through esoteric programming languages, permacomputing, fantasy computers and fantasy consoles, and whatever else shows up on the way.

Part I starts with esoteric programming languages, esolangs for short: languages made to be strange on purpose.

What makes a language esoteric

A normal programming language is designed to help people get work done. An esolang is designed for something else: a joke, a puzzle, an idea pushed as far as it goes, or a piece of art. Some remove almost everything

a language usually has and keep just enough to stay Turing complete.[1] Some hide programs inside things that don't look like programs at all, like paintings, recipes, plays or music. And some exist only to make programming as hard as possible.

Nobody writes a payroll system in them, and that's the point. Taking a language apart this way shows what a language really needs, which parts of programming are habit and which are necessity, and how little a machine needs to compute anything at all.

A short history

It began as a parody. In 1972, two Princeton students, Don Woods and James M. Lyon, wrote INTERCAL, a language meant to be unlike every language they knew.[2] Its best-known rule is about manners: statements can start with `PLEASE`, and the compiler rejects a program that says it too rarely, for being impolite, or too often, for being excessive. For years INTERCAL existed mostly as a paper manual. In 1990, C-INTERCAL brought it to Unix, and the revival got a lot of people interested in designing strange languages on purpose.

1993 was a busy year. Wouter van Oortmerssen made FALSE, a tiny stack language with a compiler of 1024 bytes. That inspired Urban Müller, a Swiss student, to go smaller: Brainfuck, eight commands, with a compiler for the Amiga that took 296 bytes and later 240. The same year Chris Pressey made Befunge, also for the Amiga, where the program is a two-dimensional grid and arrows steer the instruction pointer around it. It was designed to be as hard to compile as possible.[3] This is a whole Befunge program:

```
"!dlroW ,olleH">:#,_@
```

The quotes push the text onto a stack, backwards, so it comes out in the right order. Then `>:#,_` is the loop. While the top of the stack isn't zero, the pointer bounces back and forth through it, printing one character per turn, and when the stack runs empty, `_` sends the pointer right instead, onto `@`, which ends the program. It prints `Hello, World!`.

In 1998, Ben Olmstead made Malbolge, named after the eighth circle of hell in Dante's Inferno and built to be the hardest language possible. Its

1 A language is Turing complete when it can compute anything a computer can compute, given enough memory and time. Brainfuck is; most of the joke languages aren't.

2 The name stands for "Compiler Language With No Pronounceable Acronym".

3 The name Befunge started as a typo in an online conversation. The original Befunge-93 limits programs to a grid of 80 by 25 and isn't Turing complete; the later Funge-98 lifts the limits.

code changes itself as it runs, and what an instruction does depends on where it sits in memory. It took two years for the first Malbolge program to appear, and no person wrote it: it was found by a beam search program written by Andrew Cooke, in Lisp.

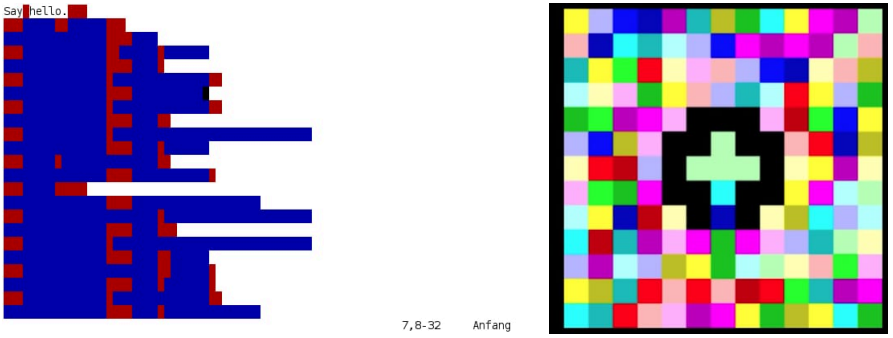
After that the field went everywhere at once. The table has some of the landmarks:

CSSV

```
---
/* cssv:title Some landmarks */
/* cssv:description A few esoteric languages and events, by year, with the
idea behind each one. */
table { width: 100%; font: 15px/1.45 source-serif-4, Georgia, serif; color:
#111; border-collapse: collapse; }
th { font: 600 11px/1.4 ibm-plex-mono, ui-monospace, monospace; text-
transform: uppercase; letter-spacing: 0.06em; color: #666; text-align: left;
padding: 0 0.7em 0.4em 0; border-bottom: 1px solid #111; }
td { padding: 0.45em 0.7em 0.45em 0; border-top: 1px solid #e6e6e6;
vertical-align: baseline; }
tbody tr:last-child td { border-bottom: 1px solid #111; }
td[data-col="year"] { font: 600 13px/1.5 ibm-plex-mono, ui-monospace,
monospace; color: #d93a2b; white-space: nowrap; }
td[data-col="language"] { font-weight: 600; }
---
year,language,the idea
1972,INTERCAL,"A parody of every language its authors knew, with a
politeness rule"
1990,C-INTERCAL,INTERCAL finally runs on Unix
1993,FALSE,A stack language with a 1024-byte compiler
1993,Brainfuck,"Eight commands, one tape, a compiler under 300 bytes"
1993,Befunge,"A program is a grid, and arrows move the pointer"
1998,Malbolge,Built to be the hardest language there is
2001,HQ9+,Four commands that each solve one classic exercise
2002,Whitespace,"Only spaces, tabs and line breaks count"
2002,Chef,Programs are cooking recipes
2005,Esolang wiki,The community gets a home
2007,LOLCODE,Programs written in lolcat speech
2009,Velato,Programs are MIDI files
```

HQ9+, by Cliff L. Biffle, is a joke about how languages get judged: `H` prints Hello World, `Q` prints the program's own source, `9` prints the lyrics of 99 Bottles of Beer, and `+` increments an accumulator that nothing can read. It solves three classic exercises in one character each and can do nothing else.

Whitespace came from Edwin Brady and Chris Morris at the University of Durham, and Slashdot reviewed it on April Fools' Day 2003. Its only commands are spaces, tabs and line breaks, and everything else in the file is a comment, which means a Whitespace program can hide inside the indentation of a program in another language. In an editor that highlights the invisible characters, it looks like this:



David Morgan-Mar wrote two of the most loved ones. In Chef, a program is a recipe, with ingredients as variables and mixing bowls as stacks, and the design goal is that recipes should work and also taste good.[4] In Piet, named after Piet Mondrian, a program is an image. The interpreter walks from one block of colour to the next, and the change in hue and lightness between two blocks decides the instruction. This painting prints “Hello world”:

The Shakespeare Programming Language, by Jon Åslund and Karl Wiberg, turns programs into plays, where the variables are characters who talk to each other on stage. LOLCODE, made in 2007 by Adam Lindsay at Lancaster University, is written the way lolcats talk:

```
HAI
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE
```

And Velato, made by Daniel Temkin in 2009, uses MIDI files as source code, so a program is also a piece of music. Temkin also runs esoteric.codes, the best place to read interviews with the people behind these languages, Ben Olmstead and Wouter van Oortmerssen among them.

Brainfuck, up close

Brainfuck is the best language to start with, because the whole machine fits in your head. There’s a tape of cells, usually 30,000 of them, each holding a byte that starts at zero. A pointer sits on one cell. Eight characters do everything, and any other character in the file is a comment:

4 Morgan-Mar points out that the sample Hello World recipe asks for 101 eggs and 111 cups of oil, which makes a lot of food for one person.

```org
| command | what it does |
|---------+--------------------------------------------------------|
| > | move the pointer one cell to the right |
| < | move the pointer one cell to the left |
| + | add 1 to the current cell |
| - | subtract 1 from the current cell |
| . | print the current cell as a character |
| , | read one character into the current cell |
| [ | if the current cell is 0, jump past the matching ] |
| ] | if the current cell isn't 0, jump back to the matching [ |
```

That's the entire language. Writing an interpreter for it is the classic first step, and it fits in under twenty lines of Python:

```python
import sys
code = [c for c in open(sys.argv[1]).read() if c in "+-<>.,[]"]
jump, stack = {}, []
for i, c in enumerate(code):
 if c == "[": stack.append(i)
 if c == "]": j = stack.pop(); jump[i], jump[j] = j, i
tape, ptr, pc = [0] * 30000, 0, 0
while pc < len(code):
 c = code[pc]
 if c == ">": ptr += 1
 elif c == "<": ptr -= 1
 elif c == "+": tape[ptr] = (tape[ptr] + 1) % 256
 elif c == "-": tape[ptr] = (tape[ptr] - 1) % 256
 elif c == ".": sys.stdout.write(chr(tape[ptr]))
 elif c == ",": tape[ptr] = ord(sys.stdin.read(1) or "\0")
 elif c == "[" and tape[ptr] == 0: pc = jump[pc]
 elif c == "]" and tape[ptr] != 0: pc = jump[pc]
 pc += 1
```

The first loop matches every `[` with its `]` before running anything, so jumps are instant. The second loop is the machine itself. Two classic programs to test it, the Hello World from Wikipedia and `cat`, which copies its input to its output:

```console
$ cat hello.bf
++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++++
..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.
$ python3 bf.py hello.bf
Hello World!
$ echo "small web" | python3 bf.py cat.bf
small web
```

`cat.bf` is five characters long: `,[.,]`. Read a character, and while it isn't zero, print it and read the next.

The trick in almost every Brainfuck program is the loop that moves numbers around. This one adds 2 and 3 and prints the result:

```
++>+++[<+>-]++++++[<++++++++>-]<.
```

It starts with 2 in the first cell and 3 in the second. The loop `[<+>-]` runs while the second cell isn't zero, moving one unit to the left each time, so the first cell ends at 5. Right before that loop starts, the tape looks like this:

DOT

```
digraph tape {
rankdir=TB;
node [fontname="Helvetica", fontsize=12, margin="0.4,0.12"];
tape [shape=record, label="<a>2|<b>3|0|0|0|..."];
ptr [shape=plaintext, label="pointer"];
ptr -> tape:b;
}
```

A digit has to be printed as its ASCII code, and 5 is character 53, so the second loop adds 6 times 8 to the first cell before `.` prints it:

CONSOLE

```
$ python3 bf.py add.bf
5
```

That second loop is multiplication: six turns, adding 8 each time. Nearly everything else in Brainfuck grows out of these same few patterns.

Where to start

The Esolang wiki has been the community's home since April 2005 and documents thousands of languages. The language list is the place to browse, and the joke language list collects the ones that can't really be programmed in, or that only rename another language, which is a useful map of ideas that have already been done.

An even smaller first target than Brainfuck is Deadfish, created by Jonathan Todd Skinner: one accumulator and four commands. `i` adds one, `d` subtracts one, `s` squares, `o` prints, and the value resets to zero whenever it reaches -1 or 256. The whole interpreter:

PYTHON

```
import sys
acc = 0
for c in sys.argv[1]:
 if c == "i": acc += 1
 elif c == "d": acc -= 1
 elif c == "s": acc *= acc
 elif c == "o": print(acc)
 if acc in (-1, 256): acc = 0
```

```
$ python3 deadfish.py iissiso
289
$ python3 deadfish.py iisssio
1
```
CONSOLE

The second program squares its way to 256, hits the reset rule, and prints 1 instead of something enormous. That little rule is the whole personality of the language.

To try languages without installing anything, Attempt This Online runs code in the browser for a long list of languages, esolangs among them. It started as a replacement for the older Try It Online, which still works but has fallen out of date.

For company, the community portal of the wiki lists the places people meet: the `#esolangs` channel on Libera.Chat, which it calls the official home for real-time discussion, an Esolangs Discord server, The Tarpit chat room on Code Golf & Coding Challenges, and, for anyone with a newsreader, the Usenet group `alt.lang.intercal`, which discusses INTERCAL and other esoteric languages to this day.

There are two good books. Strange Code, by Ronald T. Kneusel (No Starch Press, 2022), goes through the classics and walks through building two new languages from scratch. Forty-Four Esolangs, by Daniel Temkin (MIT Press, 2025), treats esolangs as art, with languages from the last fifteen years and some designed for the book.

Making your own

The easiest way to make a new language is to rename another one. This short script turns Brainfuck into words and back:

```
import sys
words = {"+": "more", "-": "less", ">": "next", "<": "back", ".": "say",
",": "hear", "[": "while", "]": "again"}
back = {w: c for c, w in words.items()}
text = open(sys.argv[2]).read()
if sys.argv[1] == "to": print(" ".join(words[c] for c in text if c in
words))
else: print("".join(back[w] for w in text.split()))
```
PYTHON

```
$ python3 words.py to add.bf
more more next more more more while back more next less again more more more
more more more while back more more more more more more more more next less
again back say
$ python3 words.py to add.bf > add.words
$ python3 words.py from add.words > add2.bf && python3 bf.py add2.bf
5
```
CONSOLE

It works, and it's also exactly the kind of language that ends up on the joke list. The wiki is full of Brainfuck with the eight symbols swapped for something else. It's a fine exercise, but a language becomes interesting when the idea changes how programs are written, not only how they're spelled.

A few questions help find that idea:

1. What's the constraint? Fewer commands, one kind of data, programs that are also something else, a memory that forgets, time running backwards. One strong limit beats five weak ones.
2. What does the machine look like? A tape, a stack, a queue, a grid, a single number, a set of rewriting rules. Draw it before writing any code.
3. What does a program look like on the page? Text, an image, sound, a recipe, a poem. This is where most of the personality lives.
4. Can it compute anything? Proving Turing completeness usually means showing how to translate Brainfuck, or another known language, into yours. If it can't, say so; the computational class of many languages is still an open question.
5. Does it already exist? Search the wiki first. Most first ideas have been done, and the variation that makes yours different is usually better than the original idea anyway.

Then the work goes in a fairly fixed order: write a short specification, write an interpreter, and write the three programs most language pages show, Hello World, `cat` and a truth machine.[5] A Python interpreter is plenty. If the language has a visual side, make the reference examples look good too.

Publishing is open to anyone. Create an account on the Esolang wiki, read the help page for language creators linked from the front page, and write the article: who made it, when, how it works, the example programs, the computational class, and a link to the interpreter. Then share it in `#esolangs` or the Discord, where people will find the bugs in the spec within the hour, which is half the fun.

What to do with them

Golf. On Code Golf & Coding Challenges, the goal is to solve a task in as few bytes as possible, and whole languages exist only for that.

5 A truth machine reads one character. If it's 0, it prints 0 and stops; if it's 1, it prints 1 forever. It's a tiny test of input, output, branching and looping in one program.

Quines, programs that print their own source, are a rite of passage in any language, and harder in an esolang. Polyglots go further: a single file that runs as two languages at once, which Whitespace makes almost too easy.

Art. Piet programs look like paintings, Velato programs are music, and people have actually baked Chef recipes. A program can be a poem, a score or a picture that also runs.

And study. Writing a Brainfuck interpreter, then a compiler that turns it into C or assembly, teaches more about how languages work than many textbooks. The language is small enough that there's nowhere to hide.

The next part of this series leaves languages behind and goes to fantasy consoles: machines that never existed, with limits chosen on purpose.

Sources

- Esoteric programming language, Wikipedia
- INTERCAL, Brainfuck, Befunge, Malbolge, Whitespace and LOLCODE, Wikipedia
- Esolang wiki, its community portal and the pages for HQ9+ and Deadfish
- Piet and Chef, by David Morgan-Mar
- esoteric.codes, by Daniel Temkin
- Images: Hello world in Piet and Whitespace in vim, Wikimedia Commons

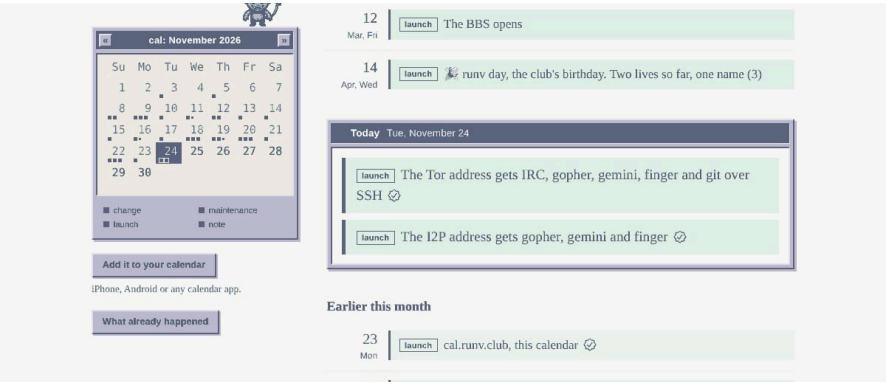


ART · NOCTEM.ART

02 RUNV · TOR · I2P 24 NOVEMBER 2026

Plumbing for runv.club

A batch of small plumbing for runv.club: a calendar built from one text file, new tricks for runvbot, an IRC network of its own with a daily channel page, and almost the whole club reachable over Tor and I2P.



Most of what keeps runv.club going is plumbing: small programs that wait for a file to change, a line in a config that opens one more door, a timer that runs once a day. This is about a batch of that plumbing, and about the club growing a few new doors.

A calendar kept in one text file

The club has a calendar now, at cal.runv.club. It's a fork of simple-calendar, a small static calendar I wrote a while ago, and the whole thing comes out of a single text file, one event per line:

```
13-11-2026 launch: runv.club on Tor [done]
24-11-2026 launch: The I2P address gets gopher, gemini and finger [done]
12-03-2027 launch: The BBS opens
14-04-2024 launch:  runv day, the club's birthday (2)
```

A date, a type, a sentence. `[done]` marks what already happened, and the `(2)` on runv day makes it come back every year with a counter. The types get colours: blue for maintenance, green for launches, lilac for changes.

The fork, at git.runv.club/runv/simple-calendar, learned a few things along the way: a dark theme that looks like `cal` in a terminal window, a page for what already happened with the newest first, the next three events at the top of the front page, and a box of its own for today, when today has something on.

Nobody runs the build by hand. A systemd path unit watches the folder, and saving is enough. It watches the folder and not the file because editors often save by writing a new file and swapping it in, and watching the file alone misses some of those saves:

```ini
# runv-cal.path
[Path]
PathChanged=/path/to/calendar/
Unit=runv-cal.service

[Install]
WantedBy=multi-user.target
```

The matching service rebuilds the site, the iCalendar file and the feed. People can subscribe on their phones and never visit the page at all.

runvbot learns to fetch things

runvbot lives in #runv and already did the small things: weather, polls, the radio, a message kept for someone until they're back. It learned a few more:

- `!img cat` brings a gif or a sticker for any word, from GIPHY.
- `!wiki lighthouse` reads the start of the Wikipedia article (`!wiki pt farol` for another language), using the opensearch API to find the title and the REST summary to get the text.
- `!search` looks for videos and songs on nine sites, and `!video 2` or `!audio 2` brings number two back as a file, with a link that lasts a day.

The bot itself runs sandboxed, with a `DynamicUser` and almost nothing it can write. It doesn't download anything. It drops a small JSON request in a spool directory and goes back to chatting:

```json
{"pick": 2, "mode": "audio", "nick": "someone", "account": "someone", "to":
"#runv"}
```

A path unit notices the file and starts the worker, under its own user, with its own memory and CPU limits. When the worker is done it writes a reply in another directory, and the bot, which checks every couple of seconds, says it in the channel. The bot and the worker never share a permission. All they share is those two directories.

Every file has to fit in 200 MB, whatever its length. So the worker works out a bitrate from the duration before calling ffmpeg, and the picture shrinks as the bitrate goes down:

```python
budget = MAX_MB * 8_000_000 * 0.92 / duration # bits per second for
everything
a = 96 if budget > 600_000 else 64 # audio, kbps
v = int(min(2_400_000, budget - a * 1000) / 1000) # video, kbps
steps = ((1400, 720), (800, 480), (450, 360), (0, 240))
height = next((h for rate, h in steps if v >= rate), 240)
# no more bits than the size of the picture can use
ceiling = {240: 600, 360: 1000, 480: 1600, 720: 2400}
v = min(v, ceiling[height])
```

A song or a short clip comes out at 720p. A two-hour film comes out at 360p or 240p and still fits. The ceiling at the end keeps short clips from getting absurd bitrates, and if a file still comes out too big, the worker tries again with 80% of the budget, then 60%.

An IRC network of its own

The club moved to its own network, irc.runv.club, running Ergo. Two neighbours came along soon after: noctem.cafe, a small OpenBSD box, and Noctem 9, a 9front server you reach with drawterm. Their people chat in #runv now.

I wanted a page like the one tilde.chat has, with the channels and how many people are in each, at runv.club/irc. It doesn't need to be live. Once a day a timer runs a short script that connects, asks the server for `LUSERS` and `LIST`, writes a static page and leaves.

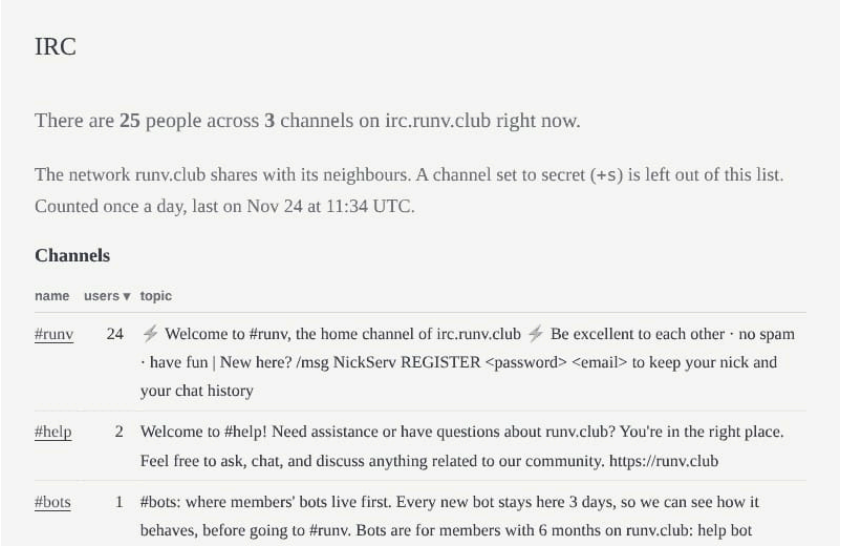
Parsing the channel list is one regular expression on each 322 reply:

```python
# :server 322 me #runv 21 :[+nt] the club's channel
chan, users, topic = re.match(r"(\S+) (\d+) :?(.*)$", rest).groups()
```

The script takes itself out of the count, since it shows up in the channels it looks at.

The whole club over Tor

runv.club already had an onion address for the website. Now almost everything else goes through the same address, `runv7hg2p4poebepvhbbstt3dcs3dqapbtq4ko4bsw5jkcepvkwmkhad.onion`:



IRC

There are **25** people across **3** channels on irc.runv.club right now.

The network runv.club shares with its neighbours. A channel set to secret (+s) is left out of this list. Counted once a day, last on Nov 24 at 11:34 UTC.

Channels

name	users ▾	topic
#runv	24	⚡ Welcome to #runv, the home channel of irc.runv.club ⚡ Be excellent to each other · no spam · have fun | New here? /msg NickServ REGISTER <password> <email> to keep your nick and your chat history
#help	2	Welcome to #help! Need assistance or have questions about runv.club? You're in the right place. Feel free to ask, chat, and discuss anything related to our community. https://runv.club
#bots	1	#bots: where members' bots live first. Every new bot stays here 3 days, so we can see how it behaves, before going to #runv. Bots are for members with 6 months on runv.club: help bot

ORG

```
| service      | port |
|--------------+------|
| the website  |   80 |
| gopher       |   70 |
| gemini       | 1965 |
| finger       |   79 |
| IRC          | 6667 |
| git over ssh | 2222 |
```

On the Tor side, each door is one `HiddenServicePort` line in `torrc`, which ties a port on the onion address to a service listening on the machine. One more line turns on Tor's proof-of-work defence, which asks clients for a small puzzle only when the service is under a flood:

```
HiddenServicePoWDefensesEnabled 1
```

Adding a port only needs a `reload`, so nobody gets dropped.

Finger and git went straight to the services that already existed. Gopher and gemini didn't, because both write their own address into what they serve. Gophernicus puts the host name in every menu line, so over Tor the first page would load and every click after it would lead back to the clear address. Molly Brown, the gemini server, goes further and refuses any request for a host other than its own, with status 53. The fix was a small copy of each, listening only on the machine, that knows the

onion address. The gopher copy is socket-activated and runs only while someone is reading:

```ini
# a gopher copy that knows the onion name
[Service]
ExecStart=/usr/sbin/gophernicus -h ONION.onion -p 70
StandardInput=socket
ProtectSystem=strict
ProtectHome=yes
```

The gemini copy has a certificate of its own for the onion name, made once and kept for twenty years, because gemini clients pin a certificate the first time they see it. Molly Brown can't be told to listen only on localhost, so systemd does the job: `IPAddressAllow=localhost` and `IPAddressDeny=any` in a drop-in.

IRC over Tor, announced in Note IX, uses Ergo's own support for it. A listener marked with `tor: true` gets its own rules. With `require-sasl: true`, only registered nicks come in this way. Tor is the door spammers like best, and a login keeps them out without bothering anyone else. Everyone who arrives over Tor shows up as `tor-network.onion`, and Tor connections have limits of their own, apart from the regular ports, so a flood through Tor can't eat into what everyone else gets.

The shell's own SSH stayed off the onion. Through Tor every connection reaches sshd from 127.0.0.1, so the protections in front of sshd can't tell one visitor from another, and a flood could fill its queue for people coming in the normal way. Git doesn't have that problem: it has its own SSH server, inside Forgejo, on port 2222. Cloning over Tor is one block of `~/.ssh/config`:

```
Host runv-git-onion
    HostName runv7hg2p4poebepvhbbstt3dcs3dqapbtq4ko4bsw5jkcepvkwmkhad.onion
    Port 2222
    User git
    ProxyCommand nc -X 5 -x 127.0.0.1:9050 %h %p
```

And a little more of it on I2P

The club's I2P address, `jzux6h32o3tzndepnqlwwvhaoq7occkjzc2fzhozearlrj5nezuq.b32.i2p`, used to carry only the website. It now has gopher, gemini and finger too, through the same kind of small copies. In i2pd, server tunnels that share a key file share a destination, and `inport` says which port each one answers on. One address, no extra tunnels on the network:

```ini
[club]
type = http
keys = club.dat

[club-gopher]
type = server
inport = 70
keys = club.dat
```

i2pd taught me one thing on the way. A `reload` picks up new tunnels and leaves gopher and the rest alone, but the HTTP tunnels stop answering until the next restart. So for i2pd it's a restart, every time, and a few minutes of republishing.

Testing from outside

Every door got tested the way a stranger would use it: a throwaway Tor client and a throwaway i2pd, each started in a scratch directory, used once and deleted. A gopher menu has to give out the right address, a gemini capsule has to answer 20 for its own name and 53 for any other, finger has to answer, and git has to clone. It's slower than checking from the server itself, but it shows what other people see.

The docs and the wiki on runv.club have the details for anyone who wants to come in through one of these doors: the Tor page and the I2P page. ■



03 FINGER · PROTOCOLS · UNIX 25 NOVEMBER 2026

finger hi@pablomurad.com

Before profiles and status updates, there was finger someone@somewhere. The history of the protocol from Stanford in 1971, how it works in one line of text, .plan files, the CMU Coke machine, the Morris worm, and where finger still answers today.



Long before profiles, status updates and the little green dot that says someone is online, there was finger. Typing `finger someone@somewhere` in a terminal brought back who that person was, whether they were logged in right now, when they last read their mail, and a few lines they had written about themselves in a file called `.plan`.

It still works:

```
finger hi@pablomurad.com
```

A program at Stanford

Les Earnest wrote the first finger program in 1971, for WAITS, the time-sharing system at the Stanford Artificial Intelligence Laboratory. On a computer shared by many people, it answered a simple question: who else is here?

Once those computers were connected to the ARPANET, people wanted to ask the same question about other machines. In December 1977, RFC 742 described finger as a network protocol. The version still in use is RFC 1288, "The Finger User Information Protocol", written by David Zimmerman at Rutgers and published in December 1991.

The whole protocol in one paragraph

A finger server listens on TCP port 79. The client connects and sends one line of text ending in a carriage return and a line feed,[1] and the server answers with plain text and closes the connection. There's no login and no headers, and the client never gets to ask a second question on the same connection.

What goes in that one line changes the answer:

```
an empty line: list the users logged in right now
username: tell me about this user
/W username: the same, with more detail (servers are free to ignore the /W)
username@host2: please ask host2 for me, and pass on the answer
```

A finger program isn't even needed. `nc` does the job, as long as the carriage return goes along. This is runv.club, the pubnix I run, answering an empty line:

```
$ printf '\r\n' | nc runv.club 79
runv.club, a small public unix server. https://runv.club/

on now: nobody
members: 8

finger name@runv.club for someone's plan
```

The last form in that list, `user@host2` sent to host1, is forwarding: host1 connects to host2 on the client's behalf. It sounds harmless, but it lets anyone bounce queries through the server, so RFC 1288 already recommended refusing it by default in 1991. Mine doesn't forward. This is what it says to anyone who tries:

```
$ printf 'root@sdf.org\r\n' | nc pablomurad.com 79
This host publishes two Finger addresses:

hi@pablomurad.com
sky@observatorium.thatsmy.tools
```

1 Two invisible characters, `\r` and `\n`, inherited from teletypes: one sent the print head back to the start of the line, the other rolled the paper up one line. Many of the old internet protocols, from SMTP to finger, still end their lines this way.

The .plan file

Fingering a user on a Unix machine made the server read two files from their home directory and print them at the end: `.project`, usually one line about what they were working on, and `.plan`, which could be anything at all.

People used the `.plan` for what we'd later use a status, a microblog or a "now" page for: what they were doing this week, where they'd be, a joke, a quote, a drawing made of characters. Since it was just a text file, updating it meant editing the file, and everyone who fingered its owner got the new version.

The most famous `.plan` files belonged to John Carmack at id Software, who wrote development notes about the studio's games in his for years. Fans fingered him to read them, and archives of his `.plan` files, starting in 1996, are still passed around today.

Mine starts like this:

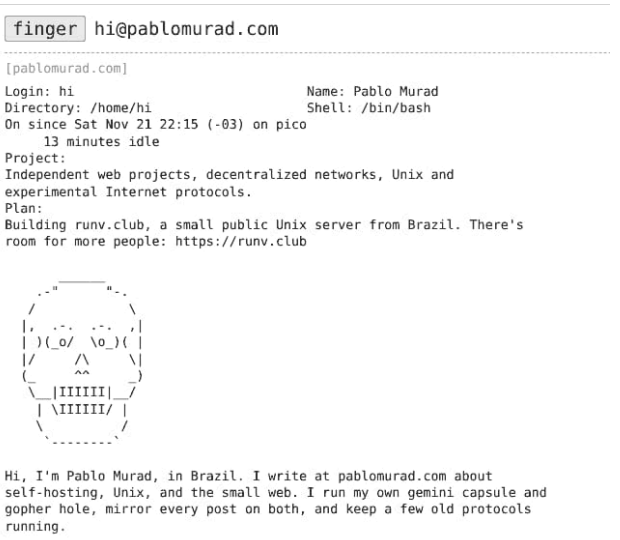
```
Login: hi Name: Pablo Murad
Directory: /home/hi Shell: /bin/bash
On since Sat Nov 21 22:15 (-03) on pico
10 minutes idle
Project:
Independent web projects, decentralized networks, Unix and experimental
Internet protocols.
Plan:
Building runv.club, a small public Unix server from Brazil.
```

After that comes a skull drawn in ASCII, a short introduction, and three sections: NOW, with my latest short posts, LATEST POSTS, with what I've published here, and ELSEWHERE, with the other ways to reach me, including `talk hi@pablomurad.com` for anyone who still has a `talk` client. The last line says when it was updated. Through a web gateway, the whole page looks like this:

The "On since" line is new. A classic finger server only says that when you're logged in to a terminal, and I almost never am, so mine now reads it from the last time the plan was rebuilt. How that works, and how a `talk` call ends up ringing my phone, is in How finger and talk work at hi@pablomurad.com.

What finger got used for, and why it faded

Finger got used for things nobody planned. In 1982, a group at Carnegie Mellon wired the Coke machine down the hall to their computer and added it to the finger program. Anyone on the ARPANET could check



whether there were bottles left, and which ones had been in long enough to be cold, before walking down the stairs. It's often called the first internet of things device.

The same openness made finger a target. In November 1988, the Morris worm, one of the first worms to spread across the internet, used a buffer overflow in `fingerd`, the finger server, as one of its ways into machines.[2]

There was also the obvious problem: finger told strangers a user's full name, when they were online, how long they had been idle and when they last read their mail. RFC 1288 itself warns that this information "may be considered sensitive". As the internet stopped being a few universities, finger stopped making sense on most machines, and over the 1990s most sites switched it off.

Where finger lives now

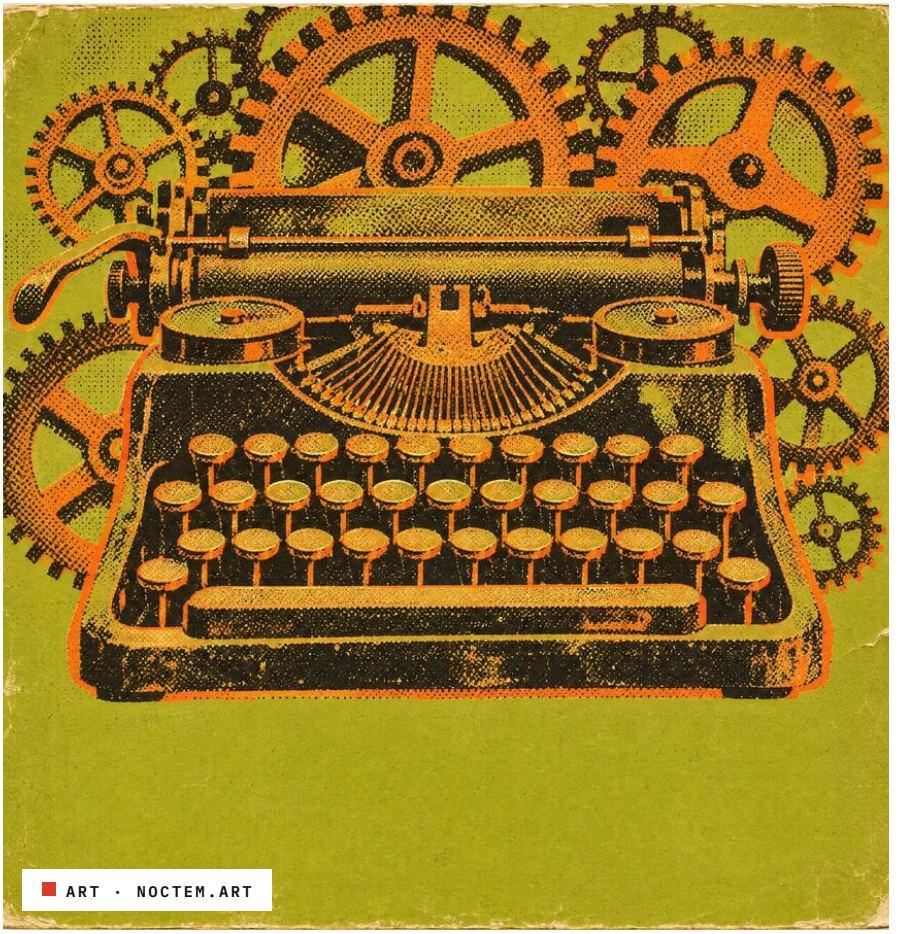
Finger survived where its original idea still fits: small shared computers where people know each other. Public access Unix systems, the pubnixes and tildes, are its natural home. I wrote about them in A short history of pubnixes and tildes. A few to try:

2 `fingerd` read the request with `gets()`, a C function that copies a line into memory without checking its length. The worm sent a line longer than the 512-byte buffer, and the extra bytes overwrote the stack with code that opened a shell.

```
finger @runv.club: who's on runv.club, and how to read someone's plan
finger @tilde.team: the users logged in to tilde.team right now
finger @sdf.org: how to get an account on SDF
finger sky@observatorium.thatsmy.tools: the Observatorium sky, my status
page, in plain text
```

Observatorium draws my servers as a night sky in text, and over finger that sky comes back as one plain block of characters.

The `finger` command comes with macOS and Windows, and on Linux it's a small package, usually called `finger`. Without it, `nc` works, as in the examples above. Anyone with an account on a pubnix can write a `.plan`, and sooner or later somebody fingers it. ■



■ ART · NOCTEM.ART

04. CRYPTOGRAPHY · SOPS · AGE 25 NOVEMBER 2026

Crypto II: My .env for everyone

Second part of the Crypto series: SOPS and age lock the values inside a .env file and leave the names readable, so it can live in a public repository. How the encryption fits together, setting it up, daily use, and what still shows.



The first part of this series was about age, which locks a whole file at once. This one is about the file that ends up holding every password in a project: the `.env`. When I wrote about what a .env file is and how to use it well, the first rule was the usual one: keep it out of Git.

SOPS bends that rule. It encrypts the values inside a config file and leaves the names in the clear, so the file still reads like a config file, still shows up in a diff like one, and can sit in a public repository. Anyone can download it. Only someone holding the right age key can read the passwords.[1]

The cover is from Poe's "The Purloined Letter", where the letter everybody is hunting for hangs in a card rack in plain view.

1 SOPS stands for Secrets OPerationS. It started at Mozilla in 2015 and was donated to the CNCF in 2023, where it lives as a sandbox project under the getsops name. Besides age it works with PGP and with the key services of AWS, Google Cloud, Azure and HuaweiCloud. The current version at the time of writing is 3.13.3.

Locked values, open names

Here's a small `.env` for an imaginary app:

```
APP_NAME=lantern
DB_HOST=localhost
DB_USER=lantern
DB_PASSWORD=correct-horse-battery
SMTP_TOKEN=tk_93f1c7a0d2
```

After SOPS, with a rule that only encrypts names ending in `_PASSWORD` or `_TOKEN`, the same file has this shape (shortened, since the real lines are long):

```
APP_NAME=lantern
DB_HOST=localhost
DB_USER=lantern
DB_PASSWORD=ENC[AES256_GCM,data:3vQx...,iv:Wm1...,tag:Hq8...,type:str]
SMTP_TOKEN=ENC[AES256_GCM,data:Zr0F...,iv:pL2...,tag:c9A...,type:str]
sops_age__list_0__map_recipient=age1zduvpw5...
sops_age__list_0__map_enc=-----BEGIN AGE ENCRYPTED FILE-----\n...
sops_encrypted_regex=_(PASSWORD|TOKEN)$
sops_lastmodified=2026-11-25T02:10:00Z
sops_mac=ENC[AES256_GCM,data:...,type:str]
sops_version=3.13.3
```

The names stay readable, and so do the values nobody needs to hide. The two secrets became `ENC[...]` blocks. The `sops_` lines at the bottom are the metadata SOPS needs to open the file again, and in YAML or JSON they go into a `sops:` section instead.

Without a rule, SOPS encrypts every value in the file and keeps only the names readable. Comments are always encrypted.

How the locks fit together

There are two layers, and age is the outer one.

1. When a file is encrypted for the first time, SOPS makes a random data key just for that file.
2. Each value is encrypted with that data key using AES-256-GCM. Every value gets its own random IV (the `iv:` part) and an authentication tag (`tag:`), and `type:` remembers whether it was text, a number or a boolean.
3. The data key itself is encrypted with age, once for each recipient, and stored in the `sops_age` lines. That's the only place age shows up.
4. Finally SOPS computes a MAC, a kind of checksum that can't be forged without the data key, over all the values, encrypted or not, and stores it encrypted in `sops_mac`.

The MAC is what makes the file tamper-proof. If someone changes `DB_HOST` by hand, adds a line or deletes one, decryption fails with a MAC mismatch. It also means the file has to be edited through SOPS, even the parts in plain text.[2]

```dot
digraph {
rankdir=LR
fontname="Helvetica"
margin="0.4,0.12"
node [shape=box, style=rounded, fontname="Helvetica", fontsize=11]
edge [fontname="Helvetica", fontsize=9, arrowsize=0.6]
key [label="your age key\n(private)"]
lock [label="sops_age lines\n(data key, locked)"]
data [label="data key\n(random, one per file)", color="#d93a2b",
fontcolor="#d93a2b"]
vals [label="ENC[...] values\nAES-256-GCM"]
mac [label="sops_mac\nchecks every value"]
key -> lock [label="opens"]
lock -> data [label="gives"]
data -> vals [label="decrypts"]
data -> mac [label="verifies"]
}
```

Compared with plain age from the first part:

```org
| | age alone | SOPS with age |
|-+-----------+---------------|
| Encrypts whole files | yes | yes |
| Keeps names and structure readable | no | yes |
| Encrypts only some values | no | yes |
| Useful diffs in Git | no | yes |
| Edit in place | decrypt, edit, encrypt again | sops edit |
| Hand secrets to a program | through a script | sops exec-env |
```

SOPS doesn't replace age. It uses age for the one job age is best at, locking a small key for a list of people, and handles the file format around it.

Setting it up

SOPS has ready binaries for Linux, macOS and Windows on its GitHub releases page, Homebrew has it as `sops`, and with Go installed `go install github.com/getsops/sops/v3/cmd/sops@latest` builds the latest one.

AN AGE KEY WHERE SOPS LOOKS

SOPS looks for age keys in a `keys.txt` file in a fixed place: `~/.config/sops/age/keys.txt` on Linux, `~/Library/Application Support/sops/age/keys.txt`

2 There's a `--mac-only-encrypted` option that computes the MAC only over the encrypted values, which leaves the plain ones free to be edited by hand.

a key for sops

```
$ age --version
1.1.1
$ mkdir -p ~/.config/sops/age
$ age-keygen -o ~/.config/sops/age/keys.txt
Public key: age1zduvpw5yrcvfy32crc9elduquw9zf8wkazgckrjdq6752gqwgyvqeu7vxy
$ stat -c "%A %s bytes" ~/.config/sops/age/keys.txt
-rw------- 189 bytes
$ age-keygen -y ~/.config/sops/age/keys.txt
age1zduvpw5yrcvfy32crc9elduquw9zf8wkazgckrjdq6752gqwgyvqeu7vxy
```

on macOS and `%AppData%\sops\age\keys.txt` on Windows. A key made with `age-keygen` straight into that file works with no extra setup:

`age-keygen` already creates the file readable only by its owner. The `age1...` line it prints is the public half, the one that goes into the config below.

A key file somewhere else can be pointed to with `SOPS_AGE_KEY_FILE`. There's also `SOPS_AGE_KEY_CMD`, which runs a command that prints the key. That opens the door to pulling the key out of the KeePassXC vault from the first part instead of keeping a `keys.txt` on disk.

A .SOPS.YAML WITH THE RULES

A `.sops.yaml` at the root of the repository tells SOPS which files to encrypt, for whom, and which values:

```
creation_rules:
- {path_regex: '\.enc\.env$', encrypted_regex: '_(PASSWORD|TOKEN)$', age:
'age1zduvpw5yrcvfy32crc9elduquw9zf8wkazgckrjdq6752gqwgyvqeu7vxy'}
```

That's ordinary YAML written in its compact, one-line form. `path_regex` picks the files, `encrypted_regex` picks the names whose values get encrypted, and `age` lists the recipients, separated by commas when there's more than one. Rules are checked from top to bottom and the first match wins. There are a few alternatives to `encrypted_regex`, like `unencrypted_suffix`, which leaves alone any name ending in `_unencrypted`. Only one of them can be used per file.

ENCRYPTING

```
$ sops encrypt app.env > app.enc.env
```
CONSOLE

SOPS reads the format from the extension. For a file that doesn't end in `.env`, `--input-type dotenv` tells it what it's looking at. From here on, `app.enc.env` goes into Git and the plain `app.env` goes into `.gitignore`, or gets deleted.

Living with it

Editing opens the decrypted file in the editor set in `$EDITOR`, waits for it to close, and encrypts again:

```
$ sops edit app.enc.env
```
CONSOLE

Reading it once, or turning it back into a plain file:

```
$ sops decrypt app.enc.env
$ sops decrypt app.enc.env > app.env
```
CONSOLE

The nicest command is `exec-env`. It decrypts the file into the environment of one program and nowhere else, so the secrets never touch the disk:

```
$ sops exec-env app.enc.env 'docker compose up -d'
```
CONSOLE

Docker Compose fills in `${DB_PASSWORD}` and the other variables in the compose file from the environment it was started with, which is exactly what `exec-env` gives it. For programs that insist on reading a file, `exec-file` hands them a temporary one instead.[3]

Git can show readable diffs too. With this line in `.gitattributes`:

```
*.enc.env diff=sopsdiffer
```

and this once per clone:

```
$ git config diff.sopsdiffer.textconv "sops decrypt"
```
CONSOLE

`git diff` decrypts both versions before comparing them, for anyone who has the key. Everyone else sees the encrypted lines change.

Adding and removing people

To give someone access, their `age1...` key goes into the `age` list in `.sops.yaml`, and then:

```
$ sops updatekeys app.enc.env
```
CONSOLE

3 By default `exec-file` passes the file through a FIFO, a special file that only exists in memory. Windows has no FIFOs, so there it needs `--no-fifo`, and then a real temporary file is written.

SOPS shows what will change and asks before doing it. This locks the existing data key for the new recipient without touching the values.

Removing someone is different. They may already have a copy of the data key, so after taking their key out of the list, the data key itself has to change:

```
$ sops updatekeys app.enc.env
$ sops rotate -i app.enc.env
```
CONSOLE

`rotate` makes a new data key and encrypts every value again. The SOPS docs recommend running it from time to time anyway.

What still shows

SOPS hides the values. Everything else is public once the file is:

- The names, the structure and the values left in plain text. A line called `STRIPE_SECRET_KEY` says a lot even encrypted.
- The length of each secret. AES-GCM doesn't pad, so the encrypted data grows with the original, and a short password still looks short.
- The history. A password committed in plain text yesterday is still in yesterday's commit, and encrypting it today doesn't help. The only fix is to change the password.
- The private key is the whole game. It needs a backup somewhere safe, and if it leaks, every secret ever encrypted with it has to be changed, not just re-encrypted.

With that in mind, the `.env` can go where everyone can see it. ■



ART · NOCTEM.ART

05 SSH · 1PASSWORD · TAILSCALE 26 NOVEMBER 2026

How I organize SSH: Tailscale for most, 1Password for the rest

How I organize SSH access on Windows after leaving Termius: Tailscale SSH for servers on the tailnet, with no keys at all, and a personal key in 1Password's SSH agent, approved with Windows Hello, for GitHub, Forgejo, tildes and the rest. Plus the plan for when something breaks.



For about two years, my SSH lived in Termius, and they were good years. The app is pleasant to use. The trouble was backups, which were always harder than they should be, and a tool that holds the keys to every door in the house shouldn't make it hard to copy the keyring. Over time I also wanted to customize more and have more control over everything, so I moved to XPipe and kept it synced between my PCs with Resilio Sync.

Today I finished the other half: organizing the keys themselves, for every service I use. It all fits in two rules.

Two rules

- **Servers on my tailnet, which is most of them:** Tailscale SSH, with no key at all.

- **Everything else** (GitHub, Forgejo, the tildes, pico and the VPSes outside Tailscale): my personal key, kept in 1Password's SSH agent and unlocked with Windows Hello. That's the public key I paste into every new account.

```dot
digraph {
rankdir=LR
fontname="Helvetica"
margin="0.4,0.12"
node [shape=box, style=rounded, fontname="Helvetica", fontsize=11]
edge [fontname="Helvetica", fontsize=9, arrowsize=0.6]
cmd [label="ssh host"]
ts [label="Tailscale SSH\ntailnet identity"]
pipe [label="Windows OpenSSH\nnamed pipe"]
op [label="1Password agent\nWindows Hello", color="#d93a2b",
fontcolor="#d93a2b"]
srv1 [label="server on\nthe tailnet"]
srv2 [label="GitHub, Forgejo,\ntildes, VPS"]
cmd -> ts [label="tailnet host"]
ts -> srv1
cmd -> pipe [label="anything else"]
pipe -> op [label="sign this?"]
op -> srv2 [label="signed"]
}
```

Rule one: Tailscale SSH

On a server, Tailscale SSH is one command, run once:

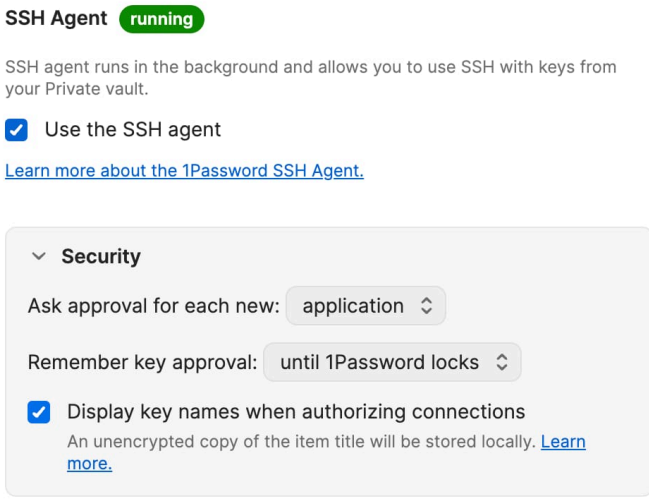
```console
$ sudo tailscale set --ssh
```

From then on, Tailscale answers SSH connections that arrive over the tailnet. The server stops asking which key I have and starts asking who I am, and Tailscale already knows, because every machine on the tailnet has its own identity.[1] There's no `authorized_keys` to keep up to date and no key to copy to a new server. It doesn't touch the server's own `sshd_config` or `authorized_keys` either, so regular SSH keeps working next to it.

Who may log in, and as which user, is decided in the tailnet's policy file. That's also where check mode lives: it asks for a fresh login with the identity provider before letting someone in, every 12 hours by default, and the period can go from a minute to a week, or every single time.

From my side nothing looks different. It's still `ssh pablo@server`, with the machine's tailnet name.

1 Tailscale builds its network on WireGuard, and every machine on the tailnet already has its own key for that. Tailscale SSH reuses that identity, so there's nothing extra to manage.



The one limit: Tailscale SSH only runs on Linux and macOS servers. Any machine can be the client.

Rule two: 1Password's agent on Windows

Windows comes with OpenSSH, and its `ssh` looks for a key agent at a fixed address, a named pipe called `\\.\pipe\openssh-ssh-agent`.[2] Normally Windows' own agent service sits there, and two agents can't sit in the same chair. So the Windows one gets up first, in a PowerShell window opened as administrator:

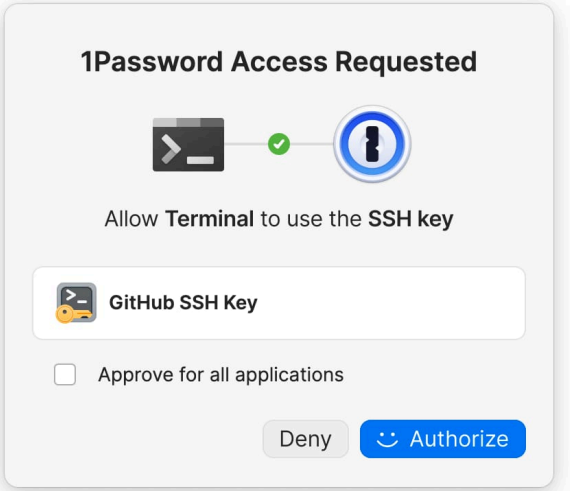
```
Stop-Service ssh-agent
Set-Service ssh-agent -StartupType Disabled
```

Then, in 1Password, Settings > Developer > Use the SSH agent. The badge at the top turns green and says the agent is running:

After that, `ssh` works the way it always did:

```
ssh vps-example
```

2 A named pipe is a channel between programs that Windows exposes with a file-like name. On Linux and macOS the same job is done by a socket file, usually pointed to by the `SSH_AUTH_SOCK` variable.



1Password pops up, shows which program wants which key, and Windows Hello asks for my fingerprint or my face, whichever gets there first:

The private key never leaves 1Password. The agent signs the login on its behalf and hands back only the signature. Both screenshots are from 1Password's own docs.

A host in `~/.ssh/config` needs nothing special, not even a path to a key file:

```
Host vps-example
HostName vps.example.org
User pablo
```

And `ssh-add -l` lists the keys the agent is offering, a quick way to check that 1Password is the one answering the door.

EVERYTHING THAT USES WINDOWS' SSH

Since the pipe belongs to the system, anything that calls Windows' `ssh` gets the same agent with no extra setup: PowerShell, XPipe, the Gram editor, and Remote-SSH in VS Code and Cursor.

Git is the exception. Git for Windows brings its own copy of `ssh` to the party, and that copy has never heard of the pipe. One line tells Git to use the system's instead:

```
git config --global core.sshCommand "C:/Windows/System32/OpenSSH/ssh.exe"
```



In my case, that line lives in the Git config of my dotfiles, so every machine gets it.

HOW OFTEN IT ASKS

A fingerprint on every single connection gets old fast. A morning of `git push` would turn into a thumb workout. So 1Password can remember an approval for a while. By default, an approval covers one program and that key until the program closes. It can be narrowed to each terminal tab, or tightened to every single request, and the memory lasts until 1Password locks, until it quits, or for a set amount of time.

While 1Password is locked, the private keys aren't in memory at all. Only the approvals are, so a remembered approval still needs the vault unlocked to work.

When something breaks

Two services carry the whole setup, and every service has a bad day eventually. For that day, I keep a local backup on a USB drive and on a server on my home network, which I reach by its local address.

Neither of them needs Tailscale or 1Password to be up. A pendrive has no login screen, no app updates and no status page, and that's exactly the job description.

What changed

The keys aren't loose files on disk anymore. Setting up a new machine means installing 1Password and Tailscale and pulling the dotfiles. And every connection outside the tailnet passes through a fingerprint first. ■

06 SSH · DOTFILES · SECURITY 26 NOVEMBER 2026

SSH, dotfiles and secrets, part 1: the PC

The longer version of my SSH post: Tailscale SSH for the tailnet and keys in 1Password for the rest, XPipe in place of Termius, dotfiles in chezmoi with signed commits, age for everything that leaves the PC, Forgejo mirrors on GitHub and licenses out of loose .txt files.



A note before you start: this is the longer, better explained version of my last SSH post, How I organize SSH. That one gives the idea in a few minutes. This one, and part 2 after it, go through every step.

The short post left a lot out. Behind it there were close to fifty hours of work from start to finish, with about eight hours of sleep in pieces along the way. Most of those hours went into the parts that didn't fit in a quick read: the dotfiles, the encryption, the mirrors and a small vault that lives on an Orange Pi.

Where I started

I had a few dozen servers and services split between Termius and a `~/.ssh` folder full of keys, one per service. The Termius side is in the last post: two good years, and backups that were always harder than they should be.

This is where things ended up on the PC, which is what part 1 covers:

1. SSH split in two: Tailscale SSH for the machines on my tailnet, and keys kept in 1Password for the rest.
2. One `~/.ssh/config`, broken into small files.
3. XPipe in place of Termius, reading that same config.
4. Dotfiles in chezmoi, in a private Git repository, with signed commits.
5. age encrypting everything that leaves the PC.
6. Private mirrors on GitHub, pushed by my Forgejo on every commit.
7. 1Password holding the licenses, recovery codes and certificates that used to live in loose .txt files.

Part 2 is about the vault: an Orange Pi Zero2 with 1 GB of RAM that stores secrets it can't read, backs itself up to the NAS and serves files with real HTTPS on the tailnet.

I'll go in the order I did it, because each step leans on the one before.

SSH in two groups

The machines on my tailnet log in through Tailscale SSH, which checks the tailnet identity and needs no key. Everything outside it (VPSes from other providers, tilde-style services and the like) uses keys that live in 1Password. 1Password works as the SSH agent, so the private keys never touch the disk and every use goes through Windows Hello. The last post shows how to turn the agent on and how to make Git use Windows' own `ssh`, so I'll skip that here.

TELLING SSH WHICH KEY TO OFFER

With several keys in the agent, SSH offers them one at a time, and the server can close the door before it gets to the right one.[1] The fix is to point `IdentityFile` at the public key. OpenSSH uses that file only to pick which key from the agent to offer, so the private half stays in 1Password.

The public key comes out of 1Password through its CLI:

```
op read "op://Personal/Example SSH key/public key" | Out-File -Encoding
ascii ~/.ssh/example.pub
```

1 OpenSSH servers count every key offered as one attempt, and by default they give up after six (`MaxAuthTries`).

A warning that cost me SSH twice: don't save that `.pub` inside `~/.ssh/1Password/`. 1Password uses that folder for its SSH Bookmarks, and when the bookmarks are turned off, it deletes the whole folder when it starts.

ONE CONFIG, SEVERAL FILES

The main config mostly pulls in the others:

```
# ~/.ssh/config
Include config.d/*.conf

Host *
ServerAliveInterval 30
AddKeysToAgent no
```

Each file in `config.d` holds one group of hosts. Hosts on the tailnet use their MagicDNS name and no key. The rest point at a public key:

```
# ~/.ssh/config.d/10-servers.conf

# on the tailnet: MagicDNS name, no IdentityFile (Tailscale SSH)
Host nas
HostName nas
User you

# outside the tailnet: a key from 1Password
Host vps
HostName vps.example.org
User you
IdentityFile ~/.ssh/example.pub
IdentitiesOnly yes
```

On each server, Tailscale SSH is `sudo tailscale set --ssh`, and the tailnet policy file decides who can log in as which user. For servers outside the tailnet, Windows has no `ssh-copy-id`, but one line of PowerShell does the same job. It asks for the server's password one last time:

```
Get-Content ~/.ssh/example.pub | ssh you@vps.example.org "umask 077; mkdir -
p ~/.ssh; cat >> ~/.ssh/authorized_keys"
```

TESTING EVERY HOST AT ONCE

With that many hosts, checking them by hand is out of the question. This PowerShell 7 script reads every `Host` line in `config.d` and tries them all in parallel. `BatchMode` makes `ssh` fail instead of asking for a password, so a broken host shows up as a failure and doesn't freeze the script:

```
#Requires -Version 7.0
$hosts = Select-String -Path ~/.ssh/config.d/*.conf -Pattern '^Host\s+(\S+)'
|
ForEach-Object { $_.Matches[0].Groups[1].Value }

$hosts | ForEach-Object -ThrottleLimit 16 -Parallel {
$null = ssh -o BatchMode=yes -o ConnectTimeout=8 $_ exit 2>&1
[pscustomobject]@{ Host = $_; OK = ($LASTEXITCODE -eq 0) }
} | Sort-Object OK, Host | Format-Table -AutoSize
```

It prints a table with each host and `True` or `False`, failures on top.

From Termius to XPipe

XPipe doesn't need servers entered one by one, because it imports `~/.ssh/config` itself. I add each file from `config.d` as an SSH config file entry, and the hosts show up inside it. User, port and key all come from the config, so the terminal, the editor and XPipe read from the same place.

Three things worth knowing before you start:

- **Hosts without `IdentityFile` show up in red.** XPipe wants a password and key choice on each one, and for the Tailscale hosts the answer is "none" in both fields. Those choices are lost every time the config file entry refreshes. I ended up with a script that copies the identity settings from one host I set by hand to the others, though doing it by hand works too.
- **XPipe's vault syncs through Git.** In the sync settings, a private repository (mine is on my Forgejo) holds the connections, scripts and identities marked for sync. Local preferences and the license stay out of it.
- **The vault's Git identity can be 1Password's agent.** In the identity used for sync, pick Password manager agent and choose the key. That way not even XPipe keeps a key on disk.

The terminal XPipe opens is Windows Terminal Preview, with copy on select turned on, because without it I couldn't copy and paste with the mouse:

```
// Windows Terminal settings.json (excerpt)
{
"copyOnSelect": true,
"firstWindowPreference": "defaultProfile",
"actions": [
{ "command": { "action": "globalSummon" }, "keys": "ctrl+alt+t" },
{ "command": "togglePaneZoom", "keys": "ctrl+shift+enter" }
]
}
```

chezmoi: every setting becomes a commit

Every config file I touched along the way (`~/.ssh`, Windows Terminal, Git, the editor, the vault scripts) lives in a private repository managed by chezmoi. It keeps a source copy of the files and knows how to apply that source to any machine, with differences per operating system.

The everyday cycle:

```
winget install twpayne.chezmoi
chezmoi init https://git.example.com/you/dotfiles.git

chezmoi add ~/.ssh/config ~/.ssh/config.d # start tracking
chezmoi re-add # changed a file directly? bring it back
chezmoi diff # what apply would change
chezmoi apply # source to machine

chezmoi git -- add -A
chezmoi git -- commit -m "ssh: one config.d file per group"
chezmoi git -- push
```

The first clone goes over HTTPS on purpose, since SSH only works after chezmoi has delivered `~/.ssh/config`. And when a script calls chezmoi, pass `--no-tty`, or it sits there waiting for a terminal that doesn't exist.

ONE FILE, TWO SYSTEMS

A file whose name ends in `.tmpl` is a template. This is my Git config, trimmed down, with commits signed by an SSH key from 1Password:

```
# dot_config/git/config.tmpl
[user]
name = Your Name
email = you@example.com
signingkey = ssh-
ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleExampleExampleExampleExample0
[gpg]
format = ssh
[gpg "ssh"]
allowedSignersFile = ~/.config/git/allowed_signers
{{- if eq .chezmoi.os "windows" }}
program = <path to op-ssh-sign.exe>
[core]
sshCommand = C:/Windows/System32/OpenSSH/ssh.exe
{{- else }}
program = <path to op-ssh-sign>
{{- end }}
[commit]
gpgsign = true
```

The `program` lines point at 1Password's signing helper, and its path depends on how 1Password was installed. 1Password can write this block for you: open the SSH key item and look for the commit signing option.

a signed commit

```
$ git log --show-signature -1
commit 32070075e08da3c70ff094502357ca9611b7d3c4
Good "git" signature for you@example.com with ED25519 key SHA256:2eYLLFSbXLKVMrriAjfZAzO68sb/ymPqAkYCvYJAo5c
Author: Pablo Murad <you@example.com>
Date:   Thu Nov 26 02:14:00 2026 -0300

    ssh: one config.d file per group
```

The `allowed_signers` file is what lets `git log --show-signature` call a signature good:

```
# dot_config/git/allowed_signers
you@example.com ssh-
ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleExampleExampleExampleExample0
```

The `.chezmoiignore` file decides what each system gets:

```
{{- if ne .chezmoi.os "windows" }}
AppData
Documents
{{- end }}
{{- if ne .chezmoi.os "linux" }}
.bashrc
.config/fastfetch
{{- end }}
meta
```

If 1Password is locked or closed, the commit fails with `failed to fill whole buffer`, an error that explains nothing. Unlock the app and try again.

PROGRAMS AND A NEW PC

The `meta/` folder in the repository, which `apply` ignores, holds what isn't a dotfile: the program list and the script I run after a fresh install. The list is a JSON file that winget reads on its own:

```
winget export -o meta/packages-winget.json
winget import -i meta/packages-winget.json --accept-package-agreements --
accept-source-agreements
```

The export picks up everything installed, so the list gets cleaned by hand afterwards.

age: encryption for everything that leaves the PC

age encrypts files to one or more public keys, and only someone holding a matching private key can open them. There's no key server and no web

```
age: a key that needs a passphrase

$ age-keygen -o key.txt
Public key: age1p3yuskta72q72kdk6w5uahfr5m8yq9uxkpa3pe7583hum62k5g6spmwp5t
$ age -p -o main.age key.txt
$ age-keygen -y key.txt > recipients.txt && shred -u key.txt
$ tar -cf - old-keys | age -R recipients.txt -o old-keys.tar.age
$ head -c 22 old-keys.tar.age; echo
age-encryption.org/v1

$ age -d -i main.age old-keys.tar.age | tar -tf -
old-keys/
old-keys/github_ed25519
old-keys/forgejo_ed25519
old-keys/tilde_ed25519
```

of trust, and the commands are short enough to remember. I prefer age over GPG for this. Here it encrypts the SD card image that goes to the cloud, an archive of old keys and every secret in the vault from part 2.

A KEY BEHIND A PASSPHRASE

The private key, which age calls an identity, gets encrypted with a passphrase right after it's created, and the open copy is deleted. age takes the encrypted file straight in `-i`: it asks for the passphrase and carries on. The public key goes into a recipients file versioned in chezmoi, because it isn't a secret.

On Windows, where there's no `shred`, the same steps look like this:

```
winget install FiloSottile.age
age-keygen -o key.txt
age -p -o main.age key.txt
age-keygen -y key.txt | Out-File -Encoding ascii ~/.config/age/
recipients.txt
Remove-Item key.txt
```

ENCRYPTING AND OPENING

```
# a disk image, compressed and encrypted in one go
xz -T0 -6 -c vault.img | age -R ~/.config/age/recipients.txt -o
vault.img.xz.age

# a whole folder
tar -cf - old-keys | age -R ~/.config/age/recipients.txt -o old-keys.tar.age

# check without extracting: list what's inside
```

```
age -d -i main.age old-keys.tar.age | tar -tf -

# open
age -d -i main.age vault.img.xz.age | xz -d > vault.img
```

Every encrypted file starts with the line `age-encryption.org/v1`, so `head -c 22` on a file is a quick way to confirm nothing went out in the clear.

The rule I follow: the private identity never goes to any server. Servers only get the public key, and they encrypt. Opening always happens on the PC.

Private mirrors: Forgejo pushes, GitHub keeps

I don't trust a single place. Forgejo runs on my server and it's where I work, and a private repository on GitHub is the second copy. I didn't want to push the dotfiles to both by hand, so Forgejo pushes a mirror on every commit.

The cleanest way is a mirror over SSH. Forgejo creates a key pair just for that mirror, and the public half goes into GitHub as a deploy key with write access to that one repository. No GitHub token is stored on the server.

In the Forgejo repository, the mirror settings take the address `git@github.com:you/dotfiles.git`, with SSH authentication and sync on commit turned on. On the GitHub side, the `gh` CLI creates the repository and adds the key:

```
gh repo create you/dotfiles --private
gh repo deploy-key add mirror.pub -R you/dotfiles -t "forgejo push mirror"
-w
```

To check it, push a commit to Forgejo only. It should show up on GitHub within seconds, with the same hash on both sides:

```
git push
git rev-parse --short HEAD
gh api repos/you/dotfiles/commits/main --jq '.sha[0:7]'
```

Never push straight to GitHub, because the mirror overwrites it. And a Forgejo shortcut while you're there: with push-to-create turned on, the first `git push` to a repository that doesn't exist yet creates it, private.

1Password: no more loose .txt files

Software licenses, account recovery codes, GPG keys, digital certificates: all of that used to live in text files in a synced folder. Now it sits in a

separate 1Password vault, with the right item type for each thing and the original files attached.

```org
| What it was | What it became in 1Password |
|---+---|
| license key in a .txt | Software License, key in the notes |
| license file (.key, .dat, .rar) | Software License with the file attached
|
| recovery codes (GitHub, MEGA, Yubico...) | Secure Note |
| GPG keys, .pfx certificates | Secure Note with the files attached |
```

THE CLI, WITHOUT A SEPARATE LOGIN

```
winget install AgileBits.1Password.CLI
# in the app: Settings, Developer, Integrate with 1Password CLI
op vault list
op vault create "Keys and Licenses"
```

With the integration on, `op` uses the app's session and asks for Windows Hello. There's no token sitting in an environment variable.

IMPORTING WITHOUT SEEING THE CONTENTS

`op item create` takes the item as JSON on standard input and the attachments as assignments, so the text goes from the file to 1Password without ever crossing the screen:

```
function New-VaultItem([string]$Title, [string]$Category, [string]$TextFile,
[string[]]$Attachments = @()) {
$item = @{ title = $Title; category = $Category; fields = @() }
if ($TextFile) {
$item.fields += @{ id = 'notesPlain'; type = 'STRING'; purpose = 'NOTES'
value = (Get-Content -LiteralPath $TextFile -Raw) }
}
# a dot in a field name starts a section: "Pro.key" has to go as "Pro\.key"
$files = @(foreach ($a in $Attachments) { "$((Split-Path $a -
Leaf).Replace('.', '\.'))[file]=$a" })
$null = $item | ConvertTo-Json -Depth 5 | op item create --vault 'Keys and
Licenses' @files
if ($LASTEXITCODE) { "ERROR $Title" } else { "ok $Title" }
}

New-VaultItem 'Program X' SOFTWARE_LICENSE 'licenses\program-x.txt'
New-VaultItem 'Program Y' SOFTWARE_LICENSE '' @('licenses\Y\license.key',
'licenses\Y\activation.pdf')
New-VaultItem 'GitHub recovery codes' SECURE_NOTE 'recovery\github.txt'
```

Two things I learned here. `@files` has to be an array even with a single attachment, which is why it's wrapped in `@(...)`. A lone string gets splatted letter by letter, and `op` refuses it. And the check is done by hash, so the value is never printed:

```
function Get-TextHash([string]$Text) {
$norm = ($Text -replace "`r`n", "`n").Trim()
[Convert]::ToHexString([Security.Cryptography.SHA256]::HashData([Text.Encoding]::UTF8.GetB
}
$note = (op item get 'Program X' --vault 'Keys and Licenses' --format json |
ConvertFrom-Json).fields |
Where-Object id -eq 'notesPlain' | ForEach-Object value
(Get-TextHash $note) -eq (Get-TextHash (Get-Content -LiteralPath
'licenses\program-x.txt' -Raw))
```

It prints `True` when the two match.

Give your functions long names. A function I had named `H` collided with the alias PowerShell already has for `Get-History`, and the error message printed the argument I passed. The argument was the secret.

Next

Part 2, out later today, is about the vault: an Orange Pi Zero2 that's ready the moment it boots, stores secrets it can't read and backs itself up to the NAS.

Links

- Tailscale SSH
- 1Password SSH agent
- XPipe
- chezmoi
- age
- Forgejo

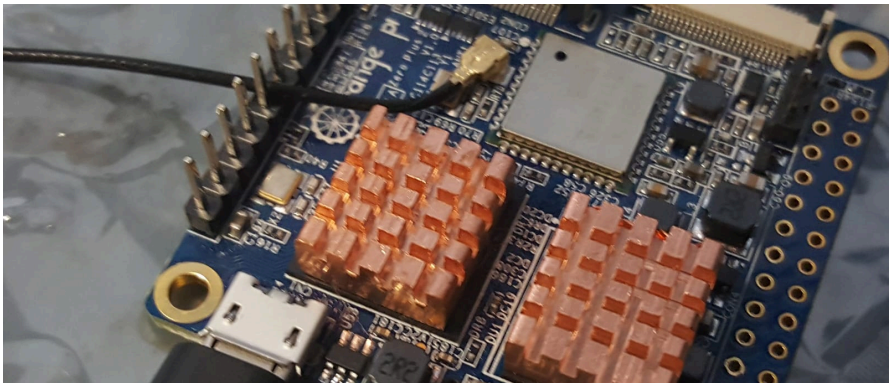


ART · NOCTEM.ART

07 HOMELAB · SELF-HOSTING · ENCRYPTION 26 NOVEMBER 2026

SSH, dotfiles and secrets, part 2: a pocket vault on an Orange Pi

Part two: an Orange Pi Zero2 turned into a pocket vault. The Armbian image is prepared in WSL so it boots ready, passage stores secrets it can't read, restic backs it up to the NAS and Caddy serves File Browser with real HTTPS on the tailnet.



A note before you start: this is the second half of the longer, better explained version of my last SSH post, How I organize SSH. Part 1 covers the PC.

Part 1 ended with everything on the PC. This part is about the vault: an Orange Pi Zero2 with 1 GB of RAM running Armbian (Debian 13, the minimal image). It stores secrets without being able to read them, pushes them to two Git hosts and backs itself up to the NAS.

```dot
digraph {
rankdir=LR
fontname="Helvetica"
margin="0.4,0.12"
node [shape=box, style=rounded, fontname="Helvetica", fontsize=11]
edge [fontname="Helvetica", fontsize=9, arrowsize=0.6]
pc [label="PC\nprivate age key"]
vault [label="Orange Pi\npublic key only", color="#d93a2b",
fontcolor="#d93a2b"]
forgejo [label="Forgejo\nprivate repo"]
```

```
github [label="GitHub\nprivate repo"]
nas [label="NAS\nrestic"]
pc -> vault [label="passage insert\nover ssh"]
vault -> forgejo [label="git push"]
vault -> github [label="git push"]
vault -> nas [label="daily backup"]
vault -> pc [label="scp the .age,\ndecrypt on the PC"]
}
```

An image that's ready before it boots

Instead of turning the board on and setting it up by hand, I prepared the image on the PC before writing it to the card. The first boot already comes up with my user, key-only login, the firewall, Tailscale and Wi-Fi. That works because the Linux inside WSL can mount the image and run its ARM programs through qemu.[1]

OPENING THE IMAGE AND GETTING INSIDE

```
#!/bin/bash
# mount.sh: in WSL (Debian), as root
# apt install qemu-user-static binfmt-support xz-utils parted e2fsprogs
set -euo pipefail
IMG=vault.img
xz -dc Armbian_*_minimal.img.xz > "$IMG"
truncate -s 4G "$IMG" # room for the new packages
L=$(losetup -f --show -P "$IMG")
parted -s "$L" resizepart 1 100%
e2fsck -fy "${L}p1" >/dev/null || true
resize2fs "${L}p1"

R=/mnt/img; mkdir -p $R; mount "${L}p1" $R
mv $R/etc/resolv.conf $R/etc/resolv.conf.orig # WSL's DNS inside the image
cp -L /etc/resolv.conf $R/etc/resolv.conf
for d in dev dev/pts proc sys; do mount --bind /$d $R/$d; done

cp personalize.sh $R/tmp/
chroot $R /bin/bash /tmp/personalize.sh

rm $R/tmp/personalize.sh $R/etc/resolv.conf
mv $R/etc/resolv.conf.orig $R/etc/resolv.conf
for d in sys proc dev/pts dev; do umount -l $R/$d; done
sync
# processes the chroot left behind (gpg-agent, dirmngr) hold the mount for a
few seconds
for i in $(seq 1 20); do umount $R 2>/dev/null && break; sleep 3; done
losetup -d "$L"
```

1 The `binfmt_misc` feature of the Linux kernel hands any ARM program to `qemu-user-static` automatically, so `chroot` into an ARM image works as if the PC were the board.

Run the whole thing as a single command. The WSL distro shuts itself down when it sits idle, and it takes the mount with it halfway through.

WHAT GOES INSIDE

`personalize.sh` runs inside the image, emulated. This is a trimmed version. The sudo setup and the firewall rules are yours to fill in:

```
#!/bin/bash
set -euo pipefail
export DEBIAN_FRONTEND=noninteractive LC_ALL=C
U=you
KEY='ssh-
ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleExampleExampleExampleExample0
you@example.com'

apt-get update -qq
apt-get install -y -qq curl git age restic ufw caddy watchdog >/dev/null
# Tailscale's own repository (trixie is Debian 13)
curl -fsSL https://pkgs.tailscale.com/stable/debian/trixie.noarmor.gpg -o /
usr/share/keyrings/tailscale-archive-keyring.gpg
curl -fsSL https://pkgs.tailscale.com/stable/debian/trixie.tailscale-
keyring.list -o /etc/apt/sources.list.d/tailscale.list
apt-get update -qq && apt-get install -y -qq tailscale >/dev/null

echo vault > /etc/hostname

# a user that logs in with a key only; root locked, no password logins
useradd -m -s /bin/bash -G sudo $U
install -d -m 700 -o $U -g $U /home/$U/.ssh
echo "$KEY" > /home/$U/.ssh/authorized_keys
chown $U:$U /home/$U/.ssh/authorized_keys; chmod 600 /home/$U/.ssh/
authorized_keys
passwd -l root >/dev/null
printf 'PasswordAuthentication no\nPermitRootLogin no\n' > /etc/ssh/
sshd_config.d/10-vault.conf
# (sudo for $U goes here, the way you like it)
rm -f /root/.not_logged_in_yet # skips Armbian's first-login wizard

# hardware watchdog: if the board hangs, it reboots itself
mkdir -p /etc/systemd/system.conf.d
printf '[Manager]\nRuntimeWatchdogSec=15\n' > /etc/systemd/system.conf.d/
watchdog.conf

# first boot: join the tailnet, destroy the auth key, turn on the firewall
cat > /usr/local/sbin/first-boot <<'EOF'
#!/bin/bash
. /etc/vault/tailscale.env 2>/dev/null || true
if [ -n "${TS_AUTHKEY:-}" ]; then
tailscale up --authkey="$TS_AUTHKEY" --ssh --hostname=vault --advertise-
tags=tag:server
fi
shred -u /etc/vault/tailscale.env 2>/dev/null || true
ufw default deny incoming
# (allow rules for your home network and the tailnet go here)
ufw --force enable
systemctl disable first-boot.service
```

```
EOF
chmod 700 /usr/local/sbin/first-boot
cat > /etc/systemd/system/first-boot.service <<'EOF'
[Unit]
Description=First boot setup
Wants=network-online.target
After=network-online.target tailscaled.service
[Service]
Type=oneshot
ExecStart=/usr/local/sbin/first-boot
[Install]
WantedBy=multi-user.target
EOF
systemctl enable first-boot.service tailscaled.service >/dev/null
apt-get clean
```

The firewall ends up accepting connections only from the home network and the tailnet. The Tailscale auth key is single use and tagged, and it goes into `/etc/vault/tailscale.env` from outside the script, only while the image is being built. The first boot uses it once and shreds it. No secret lives in the repository.

A WI-FI FALLBACK

The cable comes first, and Wi-Fi takes over if the cable goes away. netplan handles that with route metrics, and one detail held me up: the metric has to be the same for IPv4 and IPv6, or netplan rejects the file. The blog squeezes indentation, so here it is in YAML's one-line form, which netplan reads just the same:

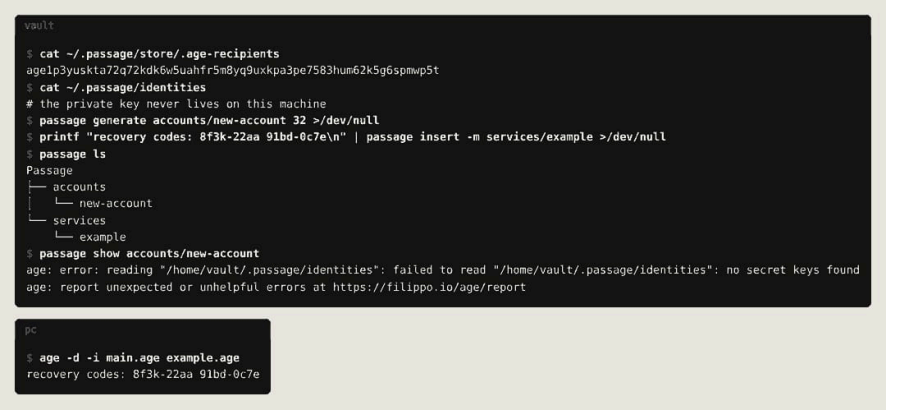
```
# /etc/netplan/20-wifi.yaml (chmod 600)
network: {version: 2, renderer: networkd, wifis: {wlan0: {dhcp4: true,
dhcp6: true, dhcp4-overrides: {route-metric: 2048}, dhcp6-overrides: {route-
metric: 2048}, regulatory-domain: BR, access-points: {"HomeNetwork": {auth:
{key-management: psk, password: "wifi-password"}}}}}}
```

Changed the Wi-Fi password with the board running? `netplan apply` alone doesn't restart `wpa_supplicant`:

```
sudo netplan apply
sudo systemctl restart netplan-wpa-wlan0.service
sudo networkctl reconfigure wlan0
```

WRITING THE CARD

`wsl --mount` doesn't take USB card readers, so the writing happens on Windows, with balenaEtcher or Rufus writing `vault.img` directly. Before the open image is deleted, it goes to the cloud encrypted with age (the commands are in part 1), and the unencrypted copy goes away with `shred -u`.



On the first boot, the vault shows up on the tailnet and `ssh vault` gets in with no key.

passage: a vault that can only encrypt

passage is the classic `pass` with age in place of GPG. Each secret is an `.age` file in a folder, and the folder is a Git repository. On the vault, I set it up to encrypt only: it knows my public key and writes secrets, but it can't read any of them, because the private key stays on the PC.

```
sudo git clone --depth 1 https://github.com/FiloSottile/passage /opt/passage
sudo ln -s /opt/passage/src/password-store.sh /usr/local/bin/passage

# only the PUBLIC key comes to the vault
mkdir -p ~/.passage/store
echo 'age1p3yuskta72q72kdk6w5uahfr5m8yq9uxkpa3pe7583hum62k5g6spmwp5t' >
~/.passage/store/.age-recipients

# passage wants an identities file even just to encrypt; here it's only
a note
printf '# the private key never lives on this machine\n' > ~/.passage/
identities
chmod 600 ~/.passage/identities

git -C ~/.passage/store init -b main
```

With `.age-recipients` at the root of the store, every insert encrypts to that public key. Reading, on the other hand, fails right there on the vault, and opens fine on the PC:

`insert -m` hands the input straight to age, so a binary file (a license `.key`, a PDF) goes in intact.

TWO REMOTES, ONE PUSH

The vault's repository has two push destinations, both private: my Forgejo and GitHub. One `git push` sends to both. The vault uses its own SSH key, added as a deploy key with write access to those two repositories and nothing else. It's the opposite of the dotfiles mirror in part 1, because here the device does the pushing.

CONSOLE

```
$ cd ~/.passage/store
$ git remote add origin git@git.example.com:you/vault.git
$ git remote set-url --add --push origin git@git.example.com:you/vault.git
$ git remote set-url --add --push origin git@github.com:you/vault.git
$ git remote -v
origin git@git.example.com:you/vault.git (fetch)
origin git@git.example.com:you/vault.git (push)
origin git@github.com:you/vault.git (push)
```

passage makes a commit on every change, and a hook pushes it. `flock` puts the pushes in a queue. Without it, an insert followed by an `rm` fired two pushes at the same time, and Forgejo ended up one commit behind.

```
#!/bin/sh
# ~/.passage/store/.git/hooks/post-commit
( sleep 3; flock /tmp/vault-push.lock git push -q origin main >/dev/null
2>&1 ) &
```

READING ON THE PC

Since the vault can't open anything, reading happens on the PC: the script pulls the `.age` file over SSH and decrypts it with the local identity.

```
# read-secret.ps1
param([Parameter(Mandatory)][string]$Name, [string]$Save)
$id = "$HOME\keys\main.age"
$tmp = New-TemporaryFile
try {
scp -q "vault:.passage/store/$Name.age" $tmp.FullName
if ($Save) { age -d -i $id -o $Save $tmp.FullName } # binary files
else { age -d -i $id $tmp.FullName }
} finally { Remove-Item $tmp.FullName -Force }
```

```
.\read-secret.ps1 accounts/new-account
.\read-secret.ps1 licenses/program.key -Save $HOME\Desktop\program.key
```

Backups to the NAS

The vault backs itself up every day to the NAS at home, over SFTP, encrypted by restic. On a Synology with DSM 7, the SSH service and the user home service both need to be on. The detail that cost me one try:



DSM's SFTP shows the user's folder as `/home`, so the repository path starts with `/home/...` and not with the real path on the volume.

```
# /etc/vault/restic.env (owned by root, chmod 600)
RESTIC_REPOSITORY='sftp:you@nas:/home/backups/vault'
RESTIC_PASSWORD='a-long-generated-password'
```

```
#!/bin/bash
# /usr/local/sbin/vault-backup
set -uo pipefail
set -a; . /etc/vault/restic.env; set +a

restic cat config >/dev/null 2>&1 || restic init
restic backup --one-file-system --exclude-caches /home/you /etc /var/lib/
filebrowser \
&& restic forget --prune <your retention flags>
```

A systemd timer runs it, with a `.service` next to it that calls the script:

```
# /etc/systemd/system/vault-backup.timer
[Timer]
OnCalendar=daily
Persistent=true
[Install]
WantedBy=timers.target
```

Then `systemctl enable --now vault-backup.timer`. The restic password goes into 1Password too, because without it the backup is worth nothing.

Real HTTPS on the tailnet

To handle files from a browser, I use File Browser behind Caddy. Caddy gets the `*.ts.net` certificate from tailscaled on its own, as long as it has permission. On the home network it uses its own internal CA.

```
echo 'TS_PERMIT_CERT_UID=caddy' | sudo tee -a /etc/default/tailscaled
sudo systemctl restart tailscaled
```

That restart drops every session that came in through Tailscale SSH, yours included. Run it while logged in through the local network address.

```
# /etc/caddy/Caddyfile
vault.example-tailnet.ts.net {
reverse_proxy 127.0.0.1:8080
}

vault.local {
tls internal
reverse_proxy 127.0.0.1:8080
}
```

```
DB=/var/lib/filebrowser/filebrowser.db
sudo install -d -o you -g you /var/lib/filebrowser /home/you/files
sudo -u you filebrowser -d $DB config init -a 127.0.0.1 -p 8080 -r /home/
you/files
sudo -u you filebrowser -d $DB users add you 'a-strong-password' --
perm.admin
```

File Browser has to be stopped for those commands, because it keeps the database locked while it runs.

One more word on Termius

There's a third reason I left Termius that didn't make it into the earlier posts: in Brazil it's expensive, and that weighed in too. Even so, it was a pleasant app for the two years I used it, and I recommend it to anyone who wants to give it a try. I'm not affiliated with Termius and I get nothing for saying this.

Links

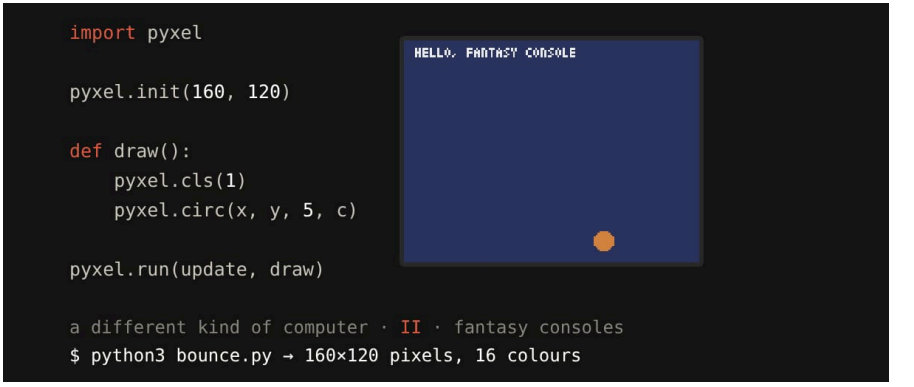
- Armbian
- passage
- age
- restic
- Caddy
- File Browser
- Tailscale SSH

$ cd ~pablo

08 ADKC · PICO-8 · GAMES 26 NOVEMBER 2026

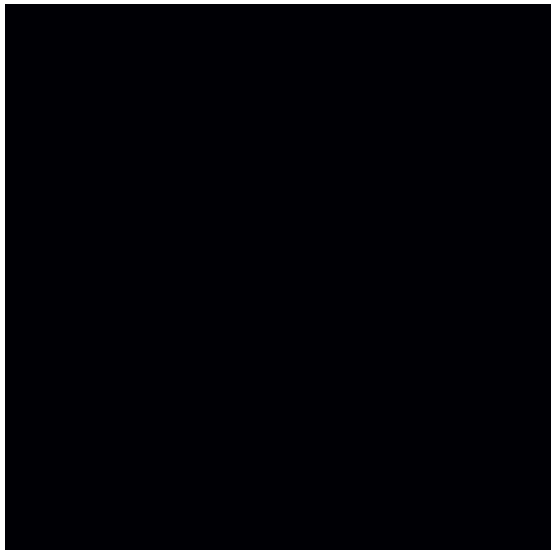
A different kind of computer II: fantasy consoles

Fantasy consoles are computers that never existed, with tiny screens, 16 colours and limits chosen on purpose. A tour of PICO-8, TIC-80, WASM-4 and Picotron, and a small Pyxel game in Python you can run today.



A fantasy console is a computer that was never built. It exists only as a program, but it behaves like hardware: it has a fixed screen size, a fixed palette, a small sound chip and a hard limit on how much code and art fit in a game. It also comes with its own tools (a code editor, a sprite editor, a map editor, a music tracker) and it saves everything into a single small file called a cartridge, which anyone else with the same machine can load and play.

The esoteric languages from earlier in this series took things away from the programmer as a joke or as an experiment. Fantasy consoles also take things away, but for a practical reason. When the screen is tiny and the palette has 16 colours, a small game becomes something one person can actually finish.



Where the name comes from

The term comes from Joseph White, known as zep, the developer behind Lexaloffle Games. His PICO-8 was first shown on 9 May 2014 at Pico Pico Cafe, still under the name PICO-BASIC, and was released in 2015. The official page describes it as “a fantasy console for making, sharing and playing tiny games and other computer programs”, and says its limitations are “carefully chosen to be fun to work with”.

The boot screen already says a lot about the idea. There is no welcome wizard and no template gallery. The machine turns on, prints its version and waits for a command, the way home computers did in the eighties. Typing `help` is the tutorial.

These are the numbers Lexaloffle publishes for PICO-8:

```
| part    | PICO-8                          |
|---------+---------------------------------|
| display | 128x128 pixels, 16 colours      |
| cart    | 32k                             |
| sound   | 4 channels                      |
| code    | P8 Lua                          |
| CPU     | 4M vm instructions per second   |
| sprites | 256 sprites of 8x8 pixels       |
| map     | 128x32 tiles                    |
| price   | $14.99                          |
```

ORG



The sound line on the official page is a little less formal than that.[1] P8 Lua is a dialect of Lua, a small scripting language that is easy to read even if you have never seen it. PICO-8 runs on Windows, macOS, Linux and the Raspberry Pi, and it can export a game as a web page or as a standalone program, so the people playing it don't need to own PICO-8.

Cartridges you can look at

PICO-8 saves cartridges as PNG images. The picture is the label of the game, and the code, sprites, map and music ride inside the pixels. A cart is something you can post on a forum like any other image, and anyone can download it, open it in PICO-8 and read the full source.[2]

That one is Celeste, which started as a PICO-8 game made in four days for a game jam.[3] Most carts live on the Lexaloffle BBS, and PICO-8 has a built-in browser called SPLORE that lists them, downloads them and runs them without leaving the machine. Reading other people's carts is one of the best ways to learn, because every game in there is also its source code.

1 Lexaloffle's spec sheet lists the sound as "4 channel chip blerps", which says something about the tone of the whole project.

2 How data fits inside an ordinary image is the subject of Hidden layers, which uses this same Celeste cart as an example.

3 That jam version later grew into the full Celeste, released for PC and consoles in 2018. The original PICO-8 cart still exists and is known today as Celeste Classic.



The other machines

PICO-8 opened the door, and other people built their own imaginary hardware with different limits.

TIC-80 calls itself "a fantasy computer for making, playing and sharing tiny games". It was created by Vadim Grigoruk (nesbox), first released in March 2017, and it is open source under the MIT license. The screen is a little wider, 240x136, still with 16 colours, and it takes up to 64KB of code in any of fourteen languages: Lua, Ruby, JavaScript, MoonScript, YueScript, Fennel, Scheme, Squirrel, Wren, WebAssembly, Janet, Python, MiniScript and Gravity. There is a paid PRO build, but the free one is complete enough to make and share games, and the site hosts a large collection of carts that you can browse from inside the console with the `surf` command.

WASM-4 goes further back in time. Its screen is 160x160 with only 4 colours, which you can redefine, and the sound has 2 pulse channels, 1 triangle and 1 noise, close to what an 8-bit console had. Memory is 64KB, carts are limited to 64KB, and a game gets 1,024 bytes of disk to save progress. The interesting choice is the language: a cart is a WebAssembly file, so you can write it in Rust, Go, AssemblyScript or anything else that compiles to WebAssembly. It also has online multiplayer with rollback netcode built in, which is unusual for something this small.

Lexaloffle kept building as well. Voxatron is "a fantasy console and collection of games made entirely out of voxels", little 3D cubes instead

of flat pixels. Picotron is what Lexaloffle calls a fantasy workstation: a whole imaginary desktop with a 480x270 screen, 64 colours and a 16-channel synthesizer. It is still in alpha.

Those are only the best known. A curated list on GitHub keeps track of dozens of these machines, and a few more are worth a quick look. LIKO-12 has a 192x128 screen, runs Lua and is open source under MIT. MicroW8 gives you 320x240 pixels with 256 colours and takes WebAssembly. nano Jammer shrinks the screen to 64x64 and runs in the browser. And Varvara, built on the uxn virtual machine, is programmed in Uxntal, a small language in the style of Forth.

The biggest difference between them is the size of the canvas. Counting the pixels on each screen shows how much room each machine gives you:

```chart
type: bar
title: Pixels on screen, in thousands
machine,thousand pixels
PICO-8,16.4
WASM-4,25.6
TIC-80,32.6
Picotron,129.6
```

And here is the short version of everything above:

```csv
---
/* cssv:title Fantasy consoles side by side */
/* cssv:description Screen, palette, language and price, as published by
each project */
table { border-collapse: collapse; width: 100%; font: 14px/1.4 system-ui,
sans-serif; }
th, td { padding: 6px 8px; border-bottom: 1px solid #e4e0d8; text-align:
left; vertical-align: top; }
th { background: #141414; color: #fff; font-weight: 600; }
td[data-col="machine"] { font-weight: 600; }
td.number, th.number { text-align: right; }
@media (max-width: 480px) { table { font-size: 12px; } th, td { padding:
5px 4px; } }
---
machine,screen,colours,code,price
PICO-8,128x128,16,P8 Lua,$14.99
TIC-80,240x136,16,14 languages,free
WASM-4,160x160,4,WebAssembly,free
Picotron,480x270,64,Lua,$19.99
Pyxel,any,16,Python,free
```

The loop inside every one of them

Underneath the different specs, all of these machines run games the same way. Your program never waits for the player. Instead, the console

calls two pieces of your code again and again, many times per second: one that updates the state of the game (positions, scores, collisions) and one that draws the current state on the screen.

```dot
digraph {
  rankdir=LR;
  node [shape=box, style=rounded, fontname="Helvetica", margin="0.4,0.12"];
  edge [fontname="Helvetica"];
  start [label="start"];
  update [label="update\nmove things, read buttons"];
  draw [label="draw\nclear screen, paint frame"];
  start -> update;
  update -> draw;
  draw -> update [label="next frame"];
}
```

In PICO-8 these are the functions `_update()` and `_draw()`. TIC-80 uses a single function, `TIC()`, which it calls 60 times per second. Pyxel asks you to pass both functions to `pyxel.run()`. Once this loop makes sense, moving from one console to another is mostly a matter of learning new function names.

A first game in Python with Pyxel

Pyxel is "a retro game engine for Python" by Takashi Kitao, free and open source under the MIT license. It follows the same rules as the consoles above: 16 colours, 4 sound channels, built-in editors for images, tilemaps and music. Since it runs on plain Python, it is the shortest way to try the idea if Python is already on your computer.

```console
$ pip install pyxel
$ python3 -c "import pyxel; print(pyxel.VERSION)"
3.0.4
$ python3 bounce.py
```

This is the whole of `bounce.py`, a ball bouncing around a 160x120 screen:

```python
import pyxel

pyxel.init(160, 120, title="bounce")
x, y, dx, dy = 20, 30, 2, 1

def update():
    global x, y, dx, dy
    x, y = x + dx, y + dy
    if x < 6 or x > 153: dx = -dx
    if y < 16 or y > 113: dy = -dy

def draw():
    pyxel.cls(1)
```




```
    pyxel.text(6, 4, "HELLO, FANTASY CONSOLE", 7)
    pyxel.circ(x, y, 5, 8 + pyxel.frame_count // 15 % 4)

pyxel.run(update, draw)
```

And this is what it produced, recorded by Pyxel itself:[4]

Going through it line by line:

- `pyxel.init(160, 120)` creates the machine. You choose the resolution, and 160x120 keeps the pixels big.
- `x, y` is where the ball is and `dx, dy` is how far it moves in each frame.
- `update()` moves the ball and flips the direction when it gets close to an edge. The numbers 6, 153, 16 and 113 are the edges minus the radius of the ball, with the top pushed down so the ball doesn't run over the text.
- `draw()` clears the screen with colour 1 (dark blue), writes the text in colour 7 (white) and draws the ball. Colours are numbers from 0 to 15, never RGB values. The ball changes colour because `frame_count // 15 % 4` counts from 0 to 3 every 60 frames, and adding 8 picks one of the colours from 8 to 11.
- `pyxel.run(update, draw)` hands both functions to the loop and never returns.

Two mistakes show up all the time when people start. The first is forgetting `global` in `update()`: without it, Python treats `x` as a new local variable

4 Pyxel has `pyxel.screenshot()` and `pyxel.screencast()`, which save the screen as a PNG or the last few seconds as an animated GIF.

and stops with `UnboundLocalError`. The second is drawing inside `update()` or changing positions inside `draw()`. It works at first, but the game gets confusing fast. Keep the rules in `update()` and the picture in `draw()`.

From here, the natural next steps are to read the arrow keys with `pyxel.btn(pyxel.KEY_LEFT)`, add a paddle, and you have the beginning of Pong. Running `pyxel edit` opens the built-in editor, where you can draw sprites and compose music for the same game.

How to start

Which machine you pick matters less than finishing something small on it. A few pointers:

- If you want the original and the biggest library of carts to learn from, PICO-8 is the one. It is paid, but a single purchase covers all platforms.
- If you want something free and open source, TIC-80 and Pyxel are both under the MIT license. TIC-80 is closer to the PICO-8 experience, with everything inside one window. Pyxel feels more like normal Python programming with a retro screen attached.
- If you already write Rust, Go or another compiled language, WASM-4 lets you keep it.

As for what to build, a game that fits on a single screen is the classic first project: Snake, Breakout, a maze, a tiny platformer with one level. But these machines are not only for games. People use them for music toys, drawing programs, small demos with visual effects and even little tools. Game jams on itch.io are a good deadline to aim for, since many of them accept PICO-8, TIC-80 and WASM-4 entries, and a jam is exactly where Celeste began.

The next part of this series leaves games behind and moves to creative coding: drawings, animations and sound made with code for their own sake, where the program is the artwork.

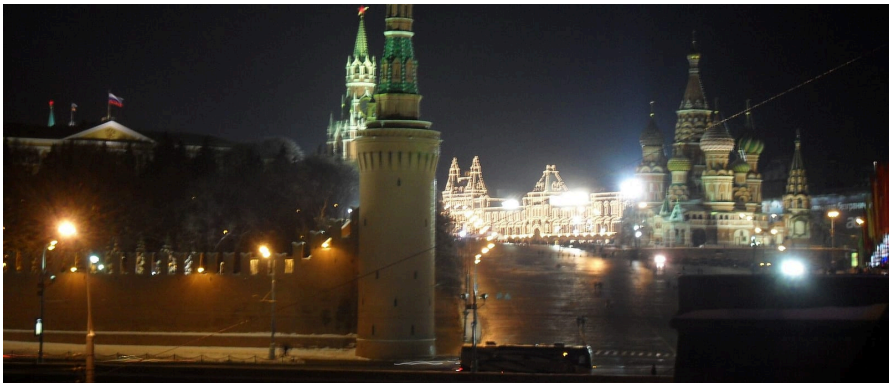
Sources

- PICO-8, Lexaloffle Games
- PICO-8 on Wikipedia
- Picotron and Voxatron, Lexaloffle Games
- TIC-80: learn and TIC-80 on Wikipedia
- WASM-4 and its documentation
- Pyxel on GitHub
- A curated list of fantasy consoles and computers on GitHub
- TIC-80 boot animation: Wikimedia Commons

09 MOVIES · POLITICS · RUSSIA 26 NOVEMBER 2026

Saw That I: The Wizard of the Kremlin

I watched The Wizard of the Kremlin, Olivier Assayas's film about the strategist behind Putin's rise. The real man behind the invented character, the real names that stay in the film, how it was made, and a line from the protagonist's father that stuck with me.



I watched *The Wizard of the Kremlin*, Olivier Assayas's film based on Giuliano da Empoli's 2022 novel, which I haven't read. Paul Dano plays Vadim Baranov, the political strategist who helps Vladimir Putin, played by Jude Law, rise to power.

What impressed me most was the way the main character is drawn. For Baranov, chaotic events are part of Russia's mission, something to twist and use to show the greatness of the country, with the idea that every evil in the world is the West's fault. Politics aside, the point of view is fascinating. Power is the main goal, and there's no right and wrong.

I found the film neutral about Putin, and neutral about left and right.

Who's who

The film plays an odd game with names. Baranov doesn't exist. He's based on Vladislav Surkov, who spent about two decades in the Kremlin as one of Putin's political strategists and is often called its "grey cardinal". Around Baranov, though, the powerful keep their own names: Putin, the oligarch Boris Berezovsky (Will Keen) and Yevgeny Prigozhin, the man who later ran the Wagner mercenaries, played by Andris Keišs.

Surkov himself knows a thing or two about borrowed names. In 2009 a novel called *Almost Zero*, a satire of corruption in Russia, came out under the pen name Natan Dubovitsky. The Russian press reported that he was widely believed to be the author. He never confirmed it, and the pen name echoes his wife's maiden name, Dubovitskaya.

Things I learned afterwards

- **A first novel that nearly won the Goncourt.** *The Wizard of the Kremlin* was Giuliano da Empoli's first novel. It won the Grand Prix du roman de l'Académie française in 2022 and missed the Prix Goncourt by the narrowest margin there is: after fourteen rounds the jury was tied 5 to 5, and its president, Didier Decoin, gave the prize to Brigitte Giraud's *Vivre vite*.
- **A screenwriter who knows Russia.** Assayas wrote the script with Emmanuel Carrère, a French writer who has spent much of his career on Russia, including *Limonov*, his book about the Russian writer and agitator Eduard Limonov.
- **Riga plays Moscow.** The film was shot in Latvia in March 2025, as many films set in Russia are. Some Latvian officials then said it served Kremlin propaganda, which is an odd fate for a film I found neutral.
- **French, but in English.** It's a French production by Gaumont, shot in English with an international cast: Dano, Law, Alicia Vikander, Tom Sturridge, Jeffrey Wright and Will Keen.
- **Assayas likes recent history.** He had done it before with *Carlos* (2010), about the terrorist Carlos the Jackal, and *Wasp Network* (2019), about Cuban spies in Miami.
- **Venice first.** It premiered in competition at the Venice Film Festival on 31 August 2025, and opened in France on 21 January 2026.

The line I took home

Baranov's father has a line that stuck with me. Translated from the Portuguese version I saw, it goes like this: rest is the benefit of those who are sick, and work is the burden of health. ■

■ ■ ■

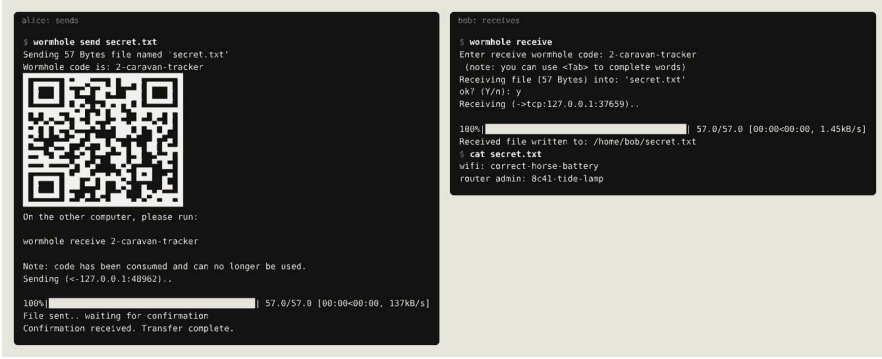
10 CRYPTOGRAPHY · MAGIC-WORMHOLE · ENCRYPTION 26 NOVEMBER 2026

Crypto III: Sending secrets through a wormhole

Third part of the Crypto series: Magic Wormhole moves a file between two computers with a short code like 7-crossover-clockwork, no accounts and no keys swapped first. How a code that small stays safe (PAKE), what the servers in the middle see, and a real transfer.



The first two parts of this series were about files at rest: age locks a whole file, and SOPS locks the values inside one. This part is about moving a secret from one computer to another, which is where most of them leak. A password pasted into a chat stays in that chat's history, on every device logged into it, and sometimes on the service's servers too.



Magic Wormhole does the move with two commands and a short code. On one computer:

```
$ wormhole send secret.txt
```
CONSOLE

On the other:

```
$ wormhole receive
```
CONSOLE

The sender gets a code like `7-crossover-clockwork`, reads it to the other person, and the file goes across, encrypted end to end. There are no accounts, no keys exchanged beforehand, and the code only works once. Brian Warner presented it at PyCon in 2016, and the current version is 0.24.0, from May 2026.

A real transfer

Here's one, with Alice sending and Bob receiving:

The run used a mailbox server and a relay on the same machine, which is why the addresses say `127.0.0.1`. With the defaults, public servers on the internet do that job, and the screens look the same.

Since version 0.22 the sender also prints the code as a QR code, and `WORMHOLE_QR=0` turns it off. On the receiving side, Tab completes the words while typing.

The code

`2-caravan-tracker` has two parts. The number is the nameplate, a channel the mailbox server hands out so both sides find each other. It isn't secret. The

words are the secret, two of them picked at random from the PGP word list, a list made to be read aloud without mistakes.[1]

Two words from that list give 16 bits, which is 65,536 possible codes. As a password, that's laughable. A zip file protected with two random words would fall to a laptop in a second, because the attacker can try every combination against the file, offline, as fast as the machine allows. Magic Wormhole is built so that attack can't happen.

Why a short code is enough

The trick is a PAKE, a password-authenticated key exchange. Magic Wormhole uses one called SPAKE2.

1. Each side mixes the code into some math and sends one message to the other, through the mailbox server.
2. If both used the same code, both end up with the same session key, long and random, the kind nobody guesses. If the codes differ, the two keys differ too.
3. Each side then sends a message encrypted with its key. If the other side can open it, they're talking to the right person. If not, the connection is closed.

The important part is what an eavesdropper gets. The SPAKE2 messages say nothing useful about the code, so recording them and trying codes later at home doesn't work. The only way to test a guess is to join the exchange live, as the other party, and each code allows exactly one try. Once someone connects, the nameplate is used up, whether the guess was right or wrong.

This is what a wrong guess looks like, also from a real run, with a third person typing a made-up code for the same nameplate:

CONSOLE

```
$ wormhole receive 4-aardvark-adroitness
ERROR: Key confirmation failed. Either you or your correspondent
typed the code wrong, or a would-be man-in-the-middle attacker guessed
incorrectly. Try sending the file again.
```

The sender got the same message at the same moment, and the code was dead. So an attacker has a 1 in 65,536 chance, one shot, and a miss shows up on the sender's screen. Even the mailbox server is in that position: if it tried to sit in the middle, it would also get one guess.

1 The PGP word list was made by Patrick Juola and Philip Zimmermann to read key fingerprints aloud. It has two lists of 256 words, one for even positions and one for odd, chosen to sound different from each other even on a bad phone line.

For more margin, `--code-length 3` adds a third word, and `--verify` prints a short string on both screens to compare over the phone before anything is sent.

The servers in the middle

There are two, and neither one can read what goes through.

The mailbox server passes the small messages: the SPAKE2 exchange first, and then messages already encrypted with the session key, including each side's IP addresses. The public one is `relay.magic-wormhole.io`.

The file itself goes by another route. The two sides first try to connect directly. When a router or firewall gets in the way, both connect to a transit relay, `transit.magic-wormhole.io` by default, which glues the two connections together and passes along bytes it can't decrypt. Either way, the data is encrypted with NaCl's SecretBox (XSalsa20 and Poly1305), using keys derived from the session key.

What the servers do see is who connected, from which IP address, when, and roughly how much data went through. Anyone who'd rather not hand that to a public server can run both pieces themselves, since they're small Python programs on PyPI:

```
$ pip install magic-wormhole-mailbox-server magic-wormhole-transit-relay
$ twist wormhole-mailbox --port=tcp:4000
$ twist transitrelay --port=tcp:4001
```
CONSOLE

Clients then point at them with `--relay-url` and `--transit-helper`, or the environment variables `WORMHOLE_RELAY_URL` and `WORMHOLE_TRANSIT_HELPER`. That's how the screenshot above was made.

Beyond files

`wormhole send` also takes a folder, which it zips on the way and unzips on arrival, and `--text` sends a short message without any file:

```
$ wormhole send --text "router admin: 8c41-tide-lamp"
```
CONSOLE

On the other side, `wormhole receive` prints the text on the screen and nothing touches the disk.

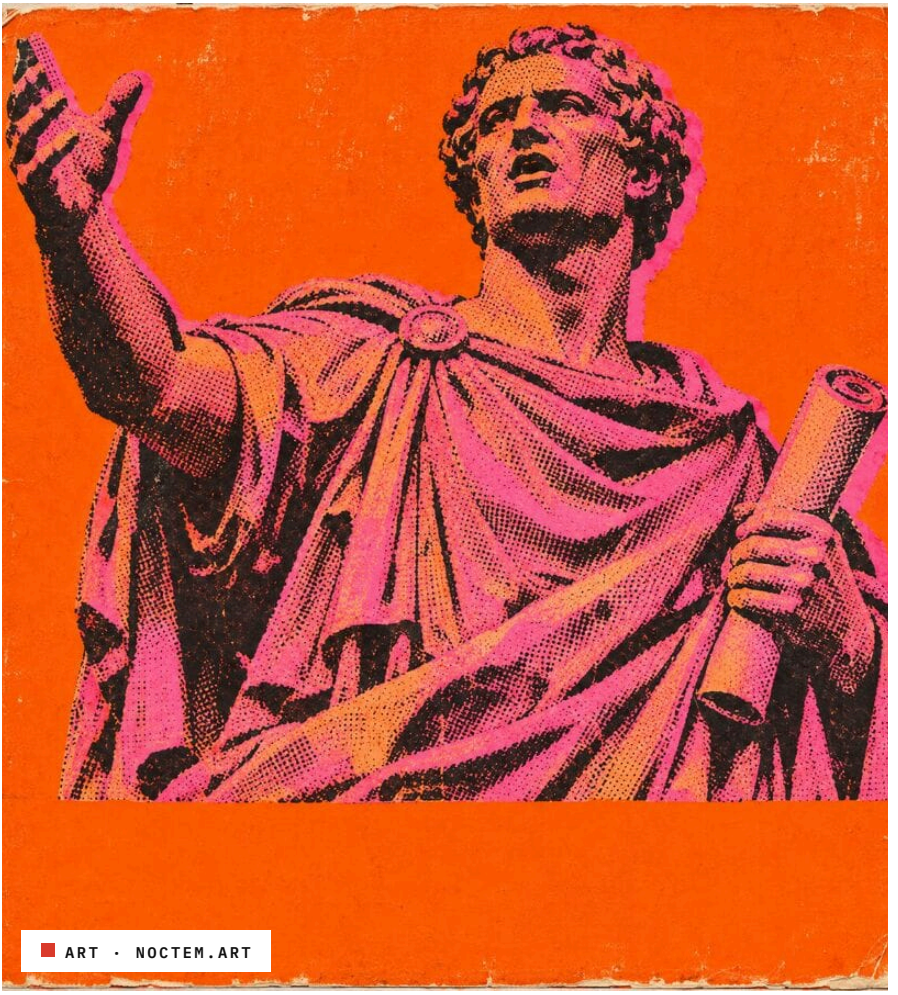
Installing

Most systems have a package: `sudo apt install magic-wormhole` on Debian and Ubuntu, `brew install magic-wormhole` on macOS, and `winget install -e --id magic-wormhole.magic-wormhole` on Windows.

`pip install magic-wormhole` gets the newest one. The version matters here: 0.23.0 and 0.24.0 both fixed security bugs in how the receiver names the files it writes, one of them a path traversal, so a package older than 0.24.0 is worth replacing.

The protocol has other clients that can talk to this one, like Warp, a graphical app for GNOME, a Wormhole app for Android, and versions written in Go and Rust.

The code is the only thing two people need to share, and it's short enough to read over the phone. ■



ART · NOCTEM.ART

11 WIKI · PROJECTS · GO 27 NOVEMBER 2026

Three ways to build the same wiki

Nyx, my personal wiki, has three versions: a Go server with Git history, a static site generator in Zig, and a single HTML file that saves itself. Each one gives a different answer to where the pages live and what you need to edit them.



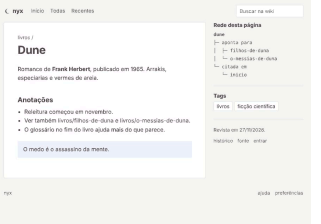
Nyx is my personal wiki, and it has three versions. Each one answers the same question in its own way: where the pages live, and what you need to edit them. All three are on my Forgejo as nyx-wiki, nyx-wiki2 and nyx-wiki3.

I made three because each one scratches a different itch. I wanted a normal wiki that I could also use from the terminal. I wanted one in Zig, because I'd been meaning to play with Zig for a long time. And I wanted one as small as it could possibly be, because I value optimization and lightness above everything else.

Nyx Wiki: a server with Git history

The first version is the one from Note X. It's written in Go. Every page is a text file with the `.wir` extension, a format of my own with a spec published in Portuguese and English. The files live in a folder that is also a Git repository, and every save becomes a commit, so the history of any page is the `git log` of its file.





One binary plays two roles. On the wiki's machine, `nyx servir` puts the pages on the web. On another computer the same program turns into a terminal client: `nyx entrar`, `nyx ver inicio`, `nyx editar livros/dune`.[1] Anyone can read, and only people with an account can write. Access control covers logins that make you wait longer after each wrong password, sessions that expire, and an API.

Pages reach the browser without JavaScript, and the server doesn't use a database. There are five themes: folha, terminal, herbario, rede and 1994. Above is terminal. Below is rede, which draws the links around the open page as a little tree in the sidebar.

The code is about 7,600 lines of Go, and the Windows executable weighs 11.5 MB.

This is the version to leave running on a server, open to the internet, with several people editing.

Nyx Wiki 2 (nyz): a static site generator in Zig

The second version is called nyz, from nyx plus Zig. It drops everything that needed a running server: logins, accounts, the editor in the browser, the API and the terminal client. You edit the `.wir` files in any text editor, and `nyz build` writes a `public/` folder of finished HTML for any static host. Links are relative, so the site works even from a subfolder.

The `.wir` format (now at version 1.2) and the five themes stay, and the themes were checked against the first version to make sure they look the same. What changed is the engineering:

- **Incremental builds.** nyz keeps the state of the last build and only renders the pages that changed, plus the ones that depend on them. It only rewrites files whose content actually changed, so an rsync upload or a commit to a pages branch carries just the difference.
- **Search.** Content pages have no JavaScript. Only the search page loads a 1.6 KB script and the index.

1 The first Nyx speaks Portuguese. Servir is serve, entrar is log in, ver is view and editar is edit.

- **Preview.** `nyz watch` rebuilds the site on every save and serves a local preview that reloads by itself.

nyz was made to be shared. Code, commands and docs are in English with a Portuguese translation, and the generated site can come out in either language. It's about 5,500 lines of Zig with no dependencies beyond the standard library,[2] and the Unicode handling and the template engine are written by hand. The binary is 941 KB. On a machine with 32 threads, 10,000 pages build from scratch in 4.5 seconds, and a build with nothing to do takes 0.14 seconds.

This is the version for people who want to write in plain files, version them however they like, and publish a site that needs nothing running on the server.

Nyx Wiki 3: the whole wiki in one HTML file

The third version goes the opposite way: everything fits in a single `nyx.html` of about 38 KB. The program, the theme and the pages, help pages included, all live inside it.

To use it, you open the file in a browser. E edits the open page, Ctrl+K searches for a page or creates one, and Ctrl+S saves. The file then writes a new copy of itself with your changes. In Chrome, Edge and Opera it saves over its own file. In Firefox and Safari it downloads an updated copy that replaces the old one.[3] Until you save, the draft is kept in the browser.

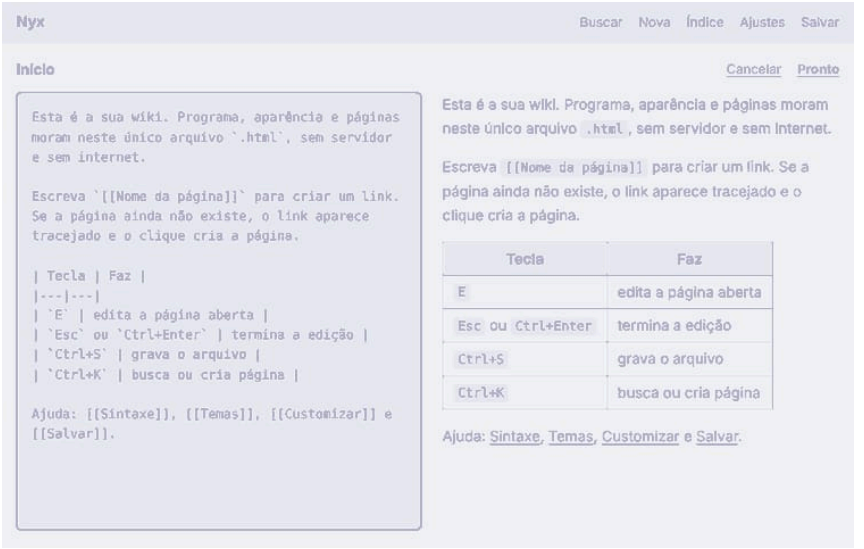
Here the format stops being `.wir`. Pages are written in Markdown with `[[wiki links]]`, and each one sits in a `<script type="text/markdown">` block inside the HTML, where you can read and edit it by hand. The wiki shows which pages point to the one you're reading, and renaming a page updates every link to it. Images, video and audio can come from any address or from a folder next to the file.

The themes changed family too. There are the four Catppuccin flavors plus an automatic light and dark mode, with 14 accent colors. Settings and extra CSS are edited inside the wiki and saved into the file, and you can export and import everything as JSON.

There's no server, no database, no build step and nothing to install. History, if there is any, depends on where you keep the file.

2 That count leaves out the Unicode tables, which a small Zig tool in the repository generates. With them, nyz has 8,850 lines of Zig.

3 The difference is the File System Access API, which lets a page write back to a file you picked. Chrome, Edge and Opera have it. Firefox and Safari don't, so there the wiki falls back to a download.

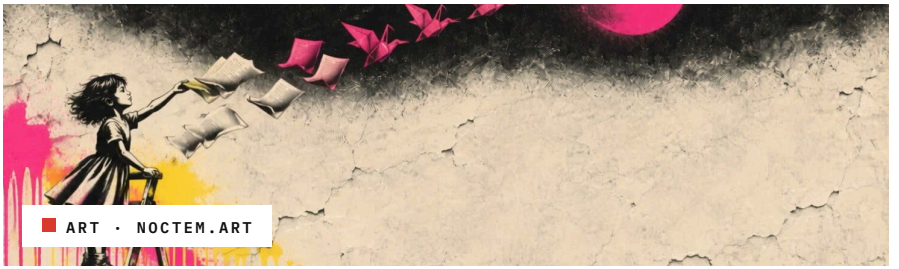


This is the version to carry on a USB stick, send by email or open on any computer.

Side by side

ORG

```
| | Nyx Wiki | nyz | Nyx Wiki 3 |
|-
| Language | Go | Zig | HTML and JavaScript |
| Pages | .wir files in Git | .wir files | Markdown inside the HTML |
| Editing | browser or terminal, with an account | any text editor | the
page itself |
| On the server | nyx servir running | any static host | nothing |
| Size | 11.5 MB (Windows) | 941 KB | about 38 KB |
| Themes | five of its own | the same five | Catppuccin, four flavors |
| History | git log | your choice | wherever the file lives |
```



12 ANDROID · SAMSUNG · ADB 28 NOVEMBER 2026

Debloating my Galaxy S25 Ultra with adb, a year late

How I cleaned up a Galaxy S25 Ultra over adb, with no root: 75 apps I never opened, 338 apps sorted into folders by use, 157 apps kept out of the background, the services that ran all day switched off, and a space theme on top.



I like Samsung phones a lot. I bought this Galaxy S25 Ultra at launch, and it's great hardware. It also comes out of the box stuffed with things I never asked for: a second app store, a second assistant, Microsoft and Meta services nobody installs on purpose, and a few "smart" features that sit in the background learning my habits. I've lived with that for about a year. Today I finally sat down and dealt with it, with no root and no custom ROM, using adb, the phone's own settings and a lot of screenshots.

```
Phone: Samsung Galaxy S25 Ultra (SM-S938B)
System: Android 17, One UI 9
Apps: 418 installed by me, 505 system packages (232 already disabled)
Tools: Android platform-tools (adb) on Windows, a bit of Python, ffmpeg
```

It started with a home screen full of icons with no labels, and everything else in a drawer sorted by brand.

Getting in

The usual first step: Developer options, USB debugging on, cable in, `adb devices`. The phone showed up, so far so good.

I wanted to see the screen on the PC, so I installed scrcpy. It froze on the first run, and right after that the phone showed up as `offline`. Two copies of adb were fighting over the same server,[1] and pointing scrcpy at a single one fixed it:

```
$ setx ADB "C:\path\to\platform-tools\adb.exe"
```
CONSOLE

The second problem was the cable. Short commands went through fine, but anything that pushed a file to the phone hung until the connection died. So I moved adb to Wi-Fi and left the cable just for charging:

```
$ adb tcpip 5555
restarting in TCP mode port: 5555
$ adb connect <phone-ip>:5555
connected to <phone-ip>:5555
```
CONSOLE

In the end I uninstalled scrcpy. Everything I needed fit in `adb exec-out screencap -p` for looking and `adb shell input tap` for touching.

Apps I never opened

Android keeps a flag for every app called `notLaunched`. It's set when the app is installed and cleared the first time it opens. Usage stats would be the obvious place to look, but on this phone they only went back about nine months, and the flag goes back to day one:

```
$ adb shell dumpsys package com.example.app | grep notLaunched
    User 0: ... installed=true ... stopped=true notLaunched=true ...
```
CONSOLE

84 apps with an icon in the drawer had never been opened. Some were installed in February 2025, the month I bought the phone. HBO Max, NordVPN, Todoist and TickTick were all in there, along with a handful of photo apps and two printer apps. I kept 9 and uninstalled the other 75, no backup:

```
$ adb shell pm uninstall com.example.app
Success
```
CONSOLE

1 scrcpy ships with its own copy of adb, and platform-tools had installed a slightly newer one. Both want to run the adb server, so each one kills the other's server and the phone gets caught in the middle. The `ADB` variable tells scrcpy to use the same adb as everything else.

`dumpsys` only knows package names, which aren't much help when deciding what to delete. To get the names you'd see on the home screen, I read the label straight out of each APK.[2]

Folders by use

The drawer had three folders: Google, Microsoft and Samsung. Sorting by brand tells you who made an app and nothing about what it's for.

I undid those three and sorted all 338 apps by what I use them for: messages, social, sound, TV and films, photos, reading, work, dev, banks, crypto, passwords, home, shopping, day to day, government and bills, games and system. The phone is in Portuguese, so the folders are too. They also have no accents, because `adb shell input text` can't type "ú". That's how the music folder ended up called Som (sound).

I stayed on One UI Home, Samsung's own launcher. There's no adb command for creating a folder, but the drawer has a selection mode with a Create folder button.

So the folders were made the way a person would make them, only faster. `uiautomator` dumps the screen as XML, with the label and position of every icon:[3]

```
$ adb shell uiautomator dump /sdcard/ui.xml
UI hierchary dumped to: /sdcard/ui.xml
```
CONSOLE

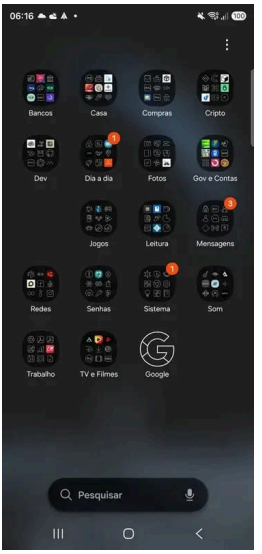
In selection mode, every icon says whether it's picked: `1Password, Selecionado` or `1Password, Não selecionado`. A small Python script read the dump, tapped every icon that belonged to the category, scrolled, read again, and pressed Create folder at the bottom of the list. The one thing missing from the dump is the popup that appears when you long-press an icon, so the script finds the Select button there by matching a cropped screenshot of it. Each folder took two or three minutes.

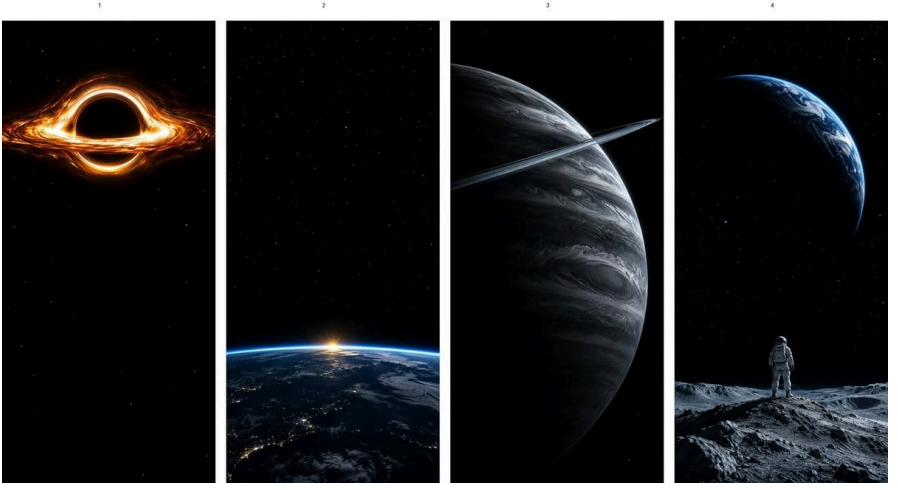
Two apps on my phone are both called Wallet, so both went into the same folder, and one of them had to be moved by hand afterward.

The home screen got the same folders through Add to Home screen, which drops each one in the first free cell. I wanted the top two rows empty for the wallpaper, so I filled them with ten copies of one folder first, added the real folders below, and then removed the copies.

2 Pulling hundreds of APKs just to read one string each would take forever over Wi-Fi. An APK is a zip file, and a zip keeps its table of contents at the end, so the script reads only the bytes it needs with `dd` over adb: the table of contents, then `AndroidManifest.xml` and `resources.arsc`. pyaxmlparser turns those two into the app's name.

3 The output really says "hierchary". That typo has been in the message for years.

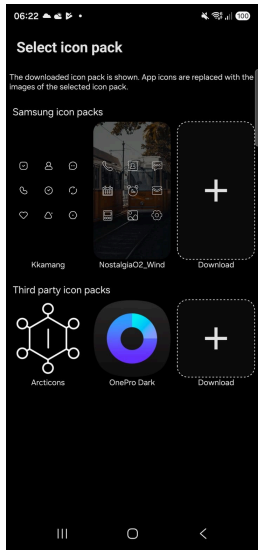




Space, minus the purple

The first wallpaper was a purple nebula, and I didn't like the purples at all. I tried four more options, all in black, graphite and steel blue:

The astronaut won. The images are 1440 x 3120, so the screen went from FHD+ to QHD+ to show them at full size. The color palette came from



Samsung's basic colors, in steel blue, because the ones it pulled from the wallpaper kept drifting toward lilac.

For icons, I bought OnePro Dark on Google Play and applied it through Theme Park, one of the Good Lock modules. Theme Park takes third-party icon packs, and apps the pack doesn't cover get the same dark tile.

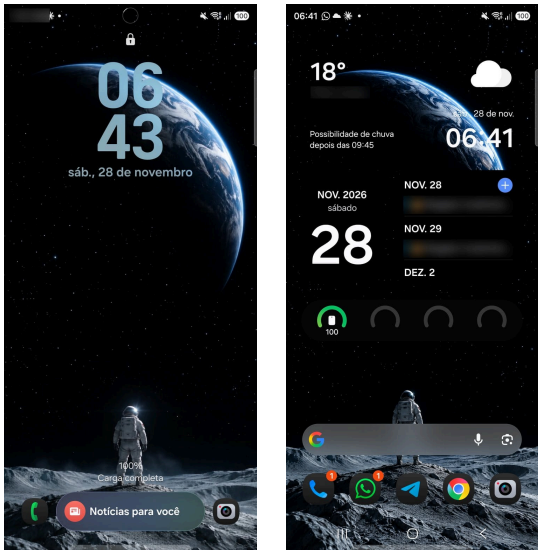
The lock screen plays a video. I turned the astronaut image into a 10-second clip, but its last frame didn't match the first, so every loop had a visible jump. One UI accepts up to 15 seconds, so ffmpeg turned the first 7.5 seconds into a loop that plays forward and then backward:

```
$ ffmpeg -i astronaut.mp4 -filter_complex "[0:v]trim=0:7.5,setpts=PTS-STARTPTS,split[a][c];[c]reverse[b];[a][b]concat=n=2:v=1[v]" -map "[v]" -an -c:v libx264 -crf 18 astronaut-loop.mp4
```
CONSOLE

One swipe to the right of the home screen there's a page with only widgets: weather and clock with the background turned off, Samsung's calendar in its big-date style, upcoming events and battery levels.

Animations went down to 0.75x. They still look like animations, only shorter:

```
$ adb shell settings put global window_animation_scale 0.75
$ adb shell settings put global transition_animation_scale 0.75
$ adb shell settings put global animator_duration_scale 0.75
```
CONSOLE



Battery and speed

The best map of what runs behind my back was 24 hours of process stats:

```
$ adb shell dumpsys procstats --hours 24
```
CONSOLE

Near the top of the list: bitchat, a Bluetooth mesh chat that scanned for other phones all day long; a message-of-the-day app; Facebook at over 600 MB of memory; TikTok at almost 800 MB; and Link to Windows with two services always on. In Samsung's own battery screen, Link to Windows came second for the day, right behind Facebook.

157 apps lost their right to run in the background: almost all of them untouched for more than 30 days, plus the heavy ones above. They still open normally, they just don't run when I'm not looking. Messaging, banking, passwords, the home cameras and the dev tools stayed off that list.

```
$ adb shell cmd appops set com.example.app RUN_ANY_IN_BACKGROUND ignore
$ adb shell am set-standby-bucket com.example.app restricted
```
CONSOLE

That's the same thing as setting an app's battery to Restricted in Settings, and it can be undone from there.

What I removed or switched off:

- bitchat and Messenger, uninstalled. Facebook stayed.



- Link to Windows, Meta App Manager, Meta Services, Meta System, Samsung's Smart Suggestions and Rubin, the personalization service. All disabled with `pm disable-user --user 0`, so they come back with `pm enable`.
- Wi-Fi and Bluetooth scanning for location, and nearby device scanning. GPS works the same.
- "Hey Google", which keeps the microphone listening all the time.

The app cache came to 34 GB, and one command cleared it:[4]

```
$ adb shell pm trim-caches 999G
```
CONSOLE

Free space went from 249 GB to 282 GB.

Last, battery protection, set to Basic: charging stops at 100% and only starts again at 95%. There's also a mode that holds the charge at 80% while I sleep and tops it up before I wake, but it needs the personalization service I had disabled a few minutes earlier.

How it ended up

The phone is fast, and every app has a place I can find without searching.

There's one small detail. In a few days I'm trading this phone for the newest model. It will come out of the box with its own Link to Windows,

4 The number is how much free space to ask the system for. 999G is more than the phone has, so it clears every app cache it's allowed to.



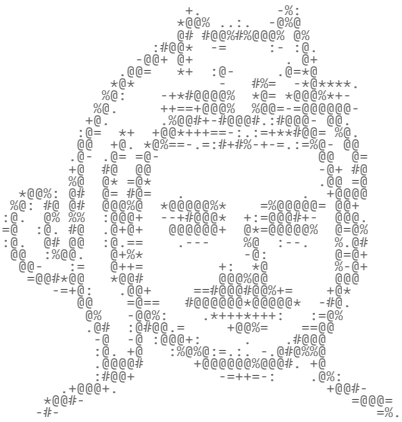
its own Meta services and its own Rubin, all very eager to meet me. The S25 Ultra has never looked better, and it gets about a week to enjoy it. ■



~pablo

A zine made of what I wrote on pablomurad.com.

04.



COLOPHON

ISSUE	Nº 04, November 2026
PUBLISHED	28 November 2026, on pablomurad.com
TYPESET	28 November 2026, A5
ESSAYS	12, each one also on the website
PLATES	Pictures between the essays are courtesy of noctem.art.
AT HOME	Print pablo-04-booklet.pdf double-sided on A4, flip on the short edge, fold the stack in half and staple the spine.

pablomurad.com/zine/04

~PABLO

Nº 04

Essays from pablomurad.com, set on paper.



