Interesting articles

- Blog: Hacking myself to prove a point
- Project: z-tokens -- Random tokens generation and related tools
- Blog: Alan Kay on “Should web browsers have stuck to being document viewers?” and a discussion of Smalltalk, NeWS and HyperCard
- OS: MenuetOS is an operating system for PC, written fully in assembly language
- Serverless Speed: Rust vs. Go, Java, and Python in AWS Lambda Functions
- execline - script language to rely entirely on chain loading
- Idenity management: Kanidm
- Rust in Perspective
- The Rise of Worse is Better
- Seizing the means of computation
- Comprehensive Python Cheatsheet
- Small Operating Systems
- A Brief History & Ethos of the Digital Garden

Blog: Hacking myself to prove a point

- https://www.macchaffee.com/blog/2023/hacking-myself/

Binary OS security (process with full user permissions OR root) is becoming a big problem. If you get execution on desktop OS as the user running it, it is all over.

Developers run lots of untrusted code during their work. While browsers sandbox untrusted code (with many security issues), developers run untrusted code directly.

OS design from 70' is not helping here.

If only we had Plan9 today, each application has it's own namespace (like docker on Linux), that can be arbitrarily nested without root privileges. Plan9 does not need root at all since there is no global namespace state to manage (all is per process); on Linux/UNIX you need root to manage namespaces (file systems/VFS, networks, processes etc.).

Another nice aspect of Plan9 in this context would be its ability to transparently run computation on separate computers over network. You can trivially run compilation on compilation farm, separated from your PC. This could be used to mitigate such vulnerability.

Project: z-tokens -- Random tokens generation and related tools

- https://github.com/volution/z-tokens

Cool project to generate random tokens and asses their binary brute force cracking resistance.

Blog: Alan Kay on “Should web browsers have stuck to being document viewers?” and a discussion of Smalltalk, NeWS and HyperCard

- https://donhopkins.medium.com/alan-kay-on-should-web-browsers-have-stuck-to-being-document-viewers-and-a-discussion-of-news-5cb92c7b3445

Funny how computer graphics, documents, web pages etc. come from printing standards and all have PostScript in common in one way or another.

Also the idea that if you can't describe something with data flexibly enough then instead of data describe it with a DSL that gets executed at the target device.

We have been moving from data to programs on the web for many years now. In printing this has happened early on and persisted to this day. Does that mean that JavaScript (or WASM) will eventually replace HTML and CSS entirely one day?

Also it explains why PostScript (and also font) processing is such a security nightmare (try finding secure PDF reader). Should we use JS/WASM sandboxes for PostScript the same way we use them in the web browsers?

OS: MenuetOS is an operating system for PC, written fully in assembly language

- https://www.menuetos.net/index.htm

Like MS-DOS and many before it. I should explore this a bit on a VM.

Serverless Speed: Rust vs. Go, Java, and Python in AWS Lambda Functions

Serverless Speed: Rust vs. Go, Java, and Python in AWS Lambda Functions

- https://blog.scanner.dev/serverless-speed-rust-vs-go-java-python-in-aws-lambda-functions/

Takeaways

- Use 1.5GB+ memory allocation for best S3 thruput
- Benchmark JSON libraries

Rust Lambda setup

```
// main.rs
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use rusoto_core::Region;
use rusoto_s3::{S3Client, S3};
use serde::{Deserialize, Serialize};
use tokio::io::{AsyncReadExt, AsyncBufReadExt};
```

```
#[derive(Deserialize)]
struct Request {
    bucket: String,
    key: String,
}

#[derive(Serialize)]
struct Response {
    req_id: String,
    msg: String,
}

async fn handle_request(event: LambdaEvent<Request>) -> Result<Response, Error> {
    let started_at = std::time::Instant::now();
    let client = S3Client::new(Region::UsWest2);

    let output = client
        .get_object(rusoto_s3::GetObjectRequest {
            bucket: bucket.to_string(),
            key: key.to_string(),
            ..Default::default()
        })
        .await?;

    let Some(body) = output.body else {
        return Err(anyhow::anyhow!("No body found in S3 response").into());
    };

    let body = body.into_async_read();
    let body = tokio::io::BufReader::new(body);
    let decoder = async_compression::tokio::bufread::ZstdDecoder::new(body);
    let reader = tokio::io::BufReader::new(decoder);

    let mut lines = reader.lines();
    let mut num_log_events = 0;
    while let Ok(Some(mut line)) = lines.next_line().await {
        let _value = unsafe {
            simd_json::to_borrowed_value(line.as_mut_str().as_bytes_mut())?
        };
        num_log_events += 1;
```

```
        if num_log_events % 1000 == 0 {
            println!("num_log_events={}", num_log_events);
        }
    }

    let msg = format!(
        "elapsed={:?} num_log_events={}",
        started_at.elapsed(),
        num_log_events
    );
    Ok(Response {
        req_id: event.context.request_id,
        msg,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .init();

    run(service_fn(handle_request)).await
}
```

Build:

```
# build.sh
cargo lambda build --release --arm64
(cd ./target/lambda/lambda_langs_test_rust/ && zip ./bootstrap.zip ./bootstrap)
```

Deploy:

```
aws lambda create-function \
--function-name lambda_langs_test_rust \
--runtime provided.al2 \
--memory-size 640  \
--architectures arm64 \
--zip-file ./bootstrap.zip \
--handler unused \
```

```
  --timeout 900 \
  --role ${LAMBDA_IAM_ROLE}
```

Run:

```
aws lambda invoke \
  --function-name lambda_langs_test_python \
  --log-type Tail \
  --cli-binary-format raw-in-base64-out \
  --payload '{"bucket": "<s3_bucket>", "key": "<s3_key>"}' \
  ./response.json \
| jq -r .LogResult | base64 --decode
```

execline - script language to rely entirely on chain loading

- https://skarnet.org/software/execline/grammar.html

- `execline` is the first script language to rely *entirely* on chain loading. An execline script is a single *argv*, made of a chain of programs designed to perform their action then `exec()` into the next one.
- The execlineb command is a *launcher*: it reads and parses a text file, converting it to an *argv*, then executes into that *argv*. It does nothing more.
- Straightforward scripts like `nice -10 echo blah` will be run just as they are, without the shell overhead. Here is what the script could look like:

  ```
  #!/command/execlineb -P
  nice -10
  echo blah
  ```

- More complex scripts will include calls to other `execline` commands, which are meant to provide some control over the process state and execution flow from inside an *argv*.

Idenity management: Kanidm

- https://github.com/kanidm/kanidm
- https://kanidm.github.io/kanidm/master/intro.html
- https://youtu.be/jeuyXhsqTBw

Written in Rust. Suse sponsored. OCID, Yubikey etc.

Rust in Perspective

> The ambition of Rust is, as I perceieve it, and whether the people driving it even knows it or not, to finish what the ALGOL committe as *primus motor* started in 1958, and what the Garmisch NATO conference concluded was necessary in 1968: to develop a language for systems programming that rely on formal logic proof, and to fulfil what ALGOL never could, what Pascal never could, and what the whole maybe-not-700 functional programming languages never could: a language that joins the disciplines of **computer science** and **software Engineering** into **ONE** discipline, where the scholars of each can solve problems together.

- https://people.kernel.org/linusw/rust-in-perspective
- https://users.rust-lang.org/t/why-are-some-people-against-the-rust-lang/93906/11

The Rise of Worse is Better

- https://dreamsongs.com/RiseOfWorseIsBetter.html

> - Simplicity -- the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
> - Correctness -- the design must be correct in all observable aspects. It is slightly better to be simple than correct.
> - Consistency -- the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
> - Completeness -- the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must be sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.
>
> Early Unix and C are examples of the use of this school of design.

> Unix and C are the ultimate computer viruses.

Seizing the means of computation

- https://www.tni.org/en/article/seizing-the-means-of-computation

> Companies will treat you how they can get away with treating you. And if they can find a way to make money by treating you like the product, they will. And if you think giving them money will make them stop, you're a sucker.

Comprehensive Python Cheatsheet

- https://gto76.github.io/python-cheatsheet/

Small Operating Systems

gemini://ew.srht.site/en/2023/20231109-re-small-operating-systems.gmi

A Brief History & Ethos of the Digital Garden

> A newly revived philosophy for publishing personal knowledge on the web

- https://maggieappleton.com/garden-history/