023-Rust-Honeypot-Dienst

Rust- und Systemaufgabe
Anforderungsniveau: Mittel

1 Ziel der Aufgabe

Sie schreiben einen schlanken Dienst in Rust, der auf mehreren Ports auf Verbindungsversuche wartet, jeden Versuch strukturiert protokolliert und die Ereignisse über eine kleine Abfrageschnittstelle auswertbar macht. Ein Auszug aus einem Mitschnitt ist weiter unten abgedruckt und dient als Grundlage für Ihre Testfälle. Geprüft wird, ob Sie nebenläufigen Netzwerkcode robust und ressourcenschonend bauen, ein auswertbares Protokollformat entwerfen und beurteilen können, welche Daten in ein Repository gehören.

Veranschlagte Bearbeitungszeit: 6–8 Stunden

- Rust mit `tokio`: mehrere Listener, nebenläufige Verbindungen
- Zeitschranken, Obergrenzen und geordnetes Herunterfahren
- Strukturiertes Ereignisprotokoll im JSON-Lines-Format
- Abfrage der Ereignisse über CLI oder HTTP
- Einheits- und Integrationstests gegen echte Sockets

2 Aufgabenbeschreibung

1. **Projekt und Konfiguration**

   Legen Sie ein Cargo-Projekt an. Die Konfiguration kommt aus einer Datei, nicht aus fest verdrahteten Werten; Umgebungsvariablen dürfen sie überschreiben.

```
# honeypot.toml
bind = "127.0.0.1"
ports = [2222, 2323, 8080]

max_verbindungen   = 256   # gleichzeitig offen
lesefrist_ms       = 3000  # Wartezeit auf Daten je Verbindung
max_bytes          = 4096  # danach wird nicht weitergelesen
protokoll          = "ereignisse.jsonl"
```

   Fehlt die Datei oder ist ein Wert unsinnig (`ports = []`, negative Fristen), bricht der Dienst mit verständlicher Meldung und einem Rückgabewert ungleich null ab – er startet nicht halb.

2. **Der lauschende Dienst**

   Für jeden konfigurierten Port läuft ein Listener. Verbindungen werden angenommen, gelesen und protokolliert. Der Dienst antwortet nicht inhaltlich.

   - Jede Verbindung wird nebenläufig bearbeitet. Eine langsame oder offen gehaltene Verbindung darf keine andere und keinen Listener blockieren.

- Es wird höchstens `max_bytes` gelesen. Wer ein Gigabyte schickt, füllt Ihnen nicht den Arbeitsspeicher.
- Nach `lesefrist_ms` ohne Daten wird die Verbindung geschlossen und das Ereignis trotzdem geschrieben – auch eine Verbindung ohne ein einziges Byte ist ein Ereignis.
- `max_verbindungen` begrenzt die gleichzeitig offenen Verbindungen, etwa mit einem Semaphor aus `tokio::sync`. Wird die Grenze erreicht, wird abgewiesen und gezählt, nicht unbegrenzt in die Warteschlange gelegt.
- `SIGINT` und `SIGTERM` führen zu einem geordneten Ende: keine neuen Verbindungen annehmen, laufende zu Ende bringen oder nach einer Frist abschneiden, Protokoll schließen – ohne Verlust bereits geschriebener Zeilen.

3. **Ereignisprotokoll**

   Jedes Ereignis ist genau eine Zeile JSON, angehängt an die Protokolldatei. Halbe Zeilen gibt es nicht: mehrere gleichzeitige Verbindungen dürfen sich nicht ineinander schreiben.

```
{"ts":"2026-01-08T02:14:07Z","quelle":"198.51.100.14","quell_port":51422,
 "ziel_port":2222,"bytes":22,"nutzlast_b64":"U1NILTIuMC1saWJzc2gyXzEuMTAuMA==",
 "dauer_ms":38,"ende":"peer_closed"}
```

   Pflichtfelder: Zeitstempel in UTC nach RFC 3339, Quelladresse und -port, Zielport, Anzahl gelesener Bytes, Nutzlast base64-kodiert, Dauer in Millisekunden und der Grund des Endes (`peer_closed`, `frist`, `limit`, `abgewiesen`). Die Nutzlast wird nie als Zeichenkette interpretiert – sie ist beliebiges Binärmaterial und darf kein `unwrap` auf ungültiges UTF-8 auslösen.

   Ältere Stände des Sensors haben die Nutzlast unkodiert in einem Feld `nutzlast_klar` abgelegt. Ihr Dienst schreibt ausschließlich `nutzlast_b64`; der Einleseweg der Abfrage muss beide Felder verstehen.

   Betriebsmeldungen (`tracing` oder `log` mit `env_logger`) gehen nach stderr, Ereignisse in die Protokolldatei – beides bleibt getrennt.

4. **Der Mitschnitt aus dem Betrieb**

   Damit Sie nicht erst tagelang Daten sammeln müssen, steht hier ein Auszug aus dem Protokoll eines Sensors. Neben den üblichen Scannern ist am 8. Januar ein falsch konfigurierter Überwachungsdienst minutenlang gegen Port 8080 gelaufen – seine Anfragen stehen genauso darin wie die der Angreifer.

```
{"ts":"2026-01-08T02:14:07Z","quelle":"198.51.100.14","quell_port":51422,"ziel_port
    ":2222,"bytes":22,"nutzlast_b64":"U1NILTIuMC1saWJzc2gyXzEuMTAuMA==","dauer_ms
    ":38,"ende":"peer_closed"}
{"ts":"2026-01-08T02:14:09Z","quelle":"198.51.100.14","quell_port":51431,"ziel_port
    ":2222,"bytes":0,"nutzlast_b64":"","dauer_ms":3001,"ende":"frist"}
{"ts":"2026-01-08T06:41:55Z","quelle":"192.0.2.77","quell_port":40118,"ziel_port
    ":8080,"bytes":66,"nutzlast_klar":"GET /.env HTTP/1.1\r\nHost: sensor.example.
    invalid\r\nUser-Agent: zgrab/0.x\r\n\r\n","dauer_ms":102,"ende":"peer_closed"}
{"ts":"2026-01-08T09:03:12Z","quelle":"203.0.113.9","quell_port":49180,"ziel_port
    ":8080,"bytes":216,"nutzlast_klar":"POST /api/v2/messwerte HTTP/1.1\r\nHost:
    messung.example.invalid\r\nAuthorization: Basic YmVudXR6ZXI6aHVudGVyMg==\r\nX-
    Api-Key: fk_test_XXXXXXXX\r\nX-Mandant: AAAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEE\r\n
    \r\n","dauer_ms":61,"ende":"peer_closed"}
{"ts":"2026-01-08T11:27:40Z","quelle":"203.0.113.201","quell_port":33214,"ziel_port
    ":2323,"bytes":27,"nutzlast_b64":"AQEAAAB0ZWxuZXQ6IHJvb3QvYWRtaW4xMjM0","
    dauer_ms":19,"ende":"peer_closed"}
```

   Die Zeile von 09:03 Uhr wiederholt sich im vollständigen Mitschnitt alle zwei Sekunden – der fehlgeleitete Überwachungsdienst macht den größten Teil des Materials aus. Bauen Sie

Ihre Testfälle auf diesem Material auf: der Einleseweg muss beide Nutzlastformen verkraften, Zeilen mit fehlenden Feldern abweisen und eine kaputte Zeile mittendrin überspringen, ohne abzubrechen. Halten Sie im README fest, was Sie mit dem Mitschnitt gemacht haben und warum.

5. **Abfrageschnittstelle**

   Die Ereignisse müssen ohne `grep`-Akrobatik auswertbar sein. Wählen Sie eine der beiden Varianten und begründen Sie die Wahl im README:

   Variante A – Unterbefehle (`clap`):

```
honeypot lauschen --config honeypot.toml
honeypot abfrage --von 2026-01-08T00:00:00Z --bis 2026-01-09T00:00:00Z \
                 --quelle 203.0.113.0/24 --ziel-port 8080 --limit 50
```

   Variante B – HTTP (`axum` oder `actix-web`), nur an `127.0.0.1` gebunden:

```
GET /ereignisse?von=...&bis=...&quelle=203.0.113.0/24&ziel_port=8080&limit=50
```

   In beiden Fällen gilt: Zeitraum, Quelladresse (einzelne Adresse oder Netzbereich, etwa mit `ipnet`) und Zielport sind kombinierbar filterbar; die Ausgabe ist begrenzt und nach Zeit absteigend sortiert. In der Listenansicht werden Nutzlasten gekürzt ausgegeben, vollständig nur auf ausdrückliche Anforderung.

6. **Tests**

   Es genügt nicht, dass etwas kompiliert und startet. Weisen Sie das Verhalten nach:

   - Einheitstests für das Einlesen einer Protokollzeile (beide Nutzlastformen, fehlende Felder, kaputte Zeile) und für die Filterlogik inklusive Netzbereich-Vergleich und Zeitraumgrenzen.
   - Einen Integrationstest, der den Dienst auf Port 0 startet, den vergebenen Port ermittelt, sich per `TcpStream` verbindet, Bytes schickt und prüft, dass genau ein Ereignis mit den erwarteten Werten im Protokoll steht. Feste Portnummern sind unzulässig – Tests müssen parallel laufen.
   - Einen Test für eine Verbindung, die nichts sendet: `ende` ist `frist`, und der Test dauert nicht länger als nötig (kurze Frist in der Testkonfiguration).

3 Anforderungen

- Rust mit `cargo`. Kein `unsafe`. Im Dienstpfad kein `unwrap` oder `expect` auf Werten, die aus dem Netz oder aus Dateien stammen – Fehler werden behandelt, nicht ausgelöst (`thiserror` oder `anyhow`).
- `cargo build` und `cargo test` laufen ohne Warnungen durch; `cargo clippy - -D warnings` ebenfalls.
- `Cargo.lock` liegt im Repository. Abhängigkeiten sind mit Version angegeben; nur tatsächlich existierende Crates.
- Kein Formatierer ohne Projektkonfiguration: wer `rustfmt` einsetzt, legt `rustfmt.toml` mit ab, sonst formatiert er bei jedem Beteiligten anders.
- Der Dienst bindet in der Voreinstellung an `127.0.0.1` und läuft ohne erhöhte Rechte; Ports unter 1024 sind nicht nötig.

- Das Repository enthält keine echten Zugangsdaten, Schlüssel, Kundenkennungen oder Kundennamen – weder in Quelltext, Tests, Testdaten noch in Konfigurationsbeispielen. Prüfen Sie das vor der Abgabe, und zwar so, dass ein nachträglich geänderter oder gelöschter Wert nicht durchrutscht. Notieren Sie im README, wie Sie geprüft haben.
- Werte für Beispiele und Tests erfinden Sie (`example.invalid`, offensichtliche Platzhalter) – man muss ihnen auf den ersten Blick ansehen, dass sie erfunden sind.

4 Abgabe

- Ein Git-Repository mit nachvollziehbarer Historie in sinnvollen Schritten, nicht ein einzelner Stand „Initial commit“ mit dem fertigen Ergebnis.
- `README.md` mit: Startanleitung, Beschreibung des Protokollformats, Erläuterung der gewählten Abfragevariante, Umgang mit dem mitgelieferten Mitschnitt, bewusst weggelassenen Punkten und dem Nachweis aus der letzten Anforderung.
- Das Projekt läuft nach dem Auschecken ohne weitere Handgriffe:

```
cargo build --release
cargo test
cargo clippy -- -D warnings
./target/release/honeypot lauschen --config honeypot.toml
```

- Im README eine kurze Sitzung zum Nachvollziehen: ein Verbindungsversuch per `nc` und die daraus entstandene Protokollzeile. Dazu eine `.gitignore` für `target/` und die Protokolldatei.

5 Bewertungskriterien

- **Robustheit unter Last** – Zeitschranken, Obergrenzen, Verbindungslimit, kein Blockieren des Listeners, geordnetes Herunterfahren.
- **Nebenläufigkeit** – keine vermischten Protokollzeilen, keine verlorenen Ereignisse.
- **Protokollformat** – vollständig, eindeutig, auswertbar; Binärdaten korrekt behandelt.
- **Abfrageschnittstelle** – kombinierbare Filter, korrekte Grenzfälle bei Zeitraum und Netzbereich, brauchbare Ausgabe.
- **Tests** – prüfen sie Verhalten oder nur, dass der Dienst startet? Wenige aussagekräftige Tests zählen mehr als viele oberflächliche.
- **Fehlerbehandlung** – verständliche Meldungen, definierte Rückgabewerte, kein Absturz bei kaputter Eingabe.
- **Sorgfalt mit den gelieferten Daten** – was landet im Repository und was nicht, und ist diese Entscheidung erkennbar getroffen worden?
- **Codequalität und Dokumentation** – Modulschnitt, Benennung, Nachvollziehbarkeit ohne Rückfrage.