Justificación Técnica – Pipeline DevSecOps

Estudiante: Wilver Vargas

Repo: https://github.com/W-Varg/Practica2 (fork de sancano22/Practica2)

Qué tiene el proyecto y cómo lo entendí

Cloné el repo y lo primero que hice fue levantar todo con docker compose para ver qué hacía antes de tocar el pipeline. El proyecto tiene 4 servicios:

- `frontend` en React/Vite — SPA con login y listado de cursos
- `users-service` en Node/Express — emite JWTs cuando el login es correcto
- `api-gateway` — valida el JWT antes de dejar pasar las peticiones
- `academic-service` — devuelve los cursos, solo accesible si venís con JWT válido

El flujo es: el usuario hace login en el frontend → `users-service` valida y emite el token → el frontend guarda el token en `sessionStorage` → todas las llamadas a cursos pasan por el `api-gateway` que verifica el token antes de redirigir al `academic-service`. Bastante estándar.

Cada servicio tiene un `/health` que devuelve un JSON con el status. Antes de agregar el pipeline verifiqué que todo respondía:

```
cd backend && docker compose up --build
curl http://localhost:3000/health   # api-gateway OK
curl http://localhost:3001/health   # users-service OK
curl http://localhost:3002/health   # academic-service OK
# frontend en http://localhost:5173
```

Todo levantó bien. Recién ahí empecé a trabajar el pipeline.

El pipeline — qué hice y por qué

El archivo está en `.github/workflows/devsecops.yml`. Ya existía una base pero le faltaban varias cosas: no cubría el frontend con SAST, no tenía ESLint como step propio, Trivy no escaneaba la imagen del frontend, y no había versionado de imágenes. Lo completé.

El orden de las etapas no es aleatorio — está pensado para fallar lo más rápido posible y en el punto más barato. No tiene sentido buildear Docker si los tests fallan, y no tiene sentido escanear con Trivy si ni siquiera construiste la imagen.

Instalación con `npm ci`

Uso `npm ci` en todos los servicios en vez de `npm install`. La diferencia importa: `npm ci` borra `node_modules` antes de instalar y usa exactamente lo que está en `package-lock.json`. Si alguien modificó el `package.json` sin actualizar el lock, el pipeline explota ahí mismo. Con `npm install` eso pasaría silenciosamente y podrías terminar con versiones distintas en CI que en local.

Tests automáticos (Jest)

Cada servicio tiene sus tests en `__tests__/`. Los corro antes del SAST porque si el código ya está roto funcionalmente, no vale la pena seguir. Jest en backend, y el frontend también tiene configurado Jest (aunque usa Vite para el build).

Si un test falla, el paso falla y GitHub Actions marca todo el job en rojo. No llega al Semgrep, no llega al Docker build. Eso está bien así.

ESLint

Lo puse como step separado del `npm test`. Podría haberlo metido dentro del mismo comando pero lo separé para que en los logs de Actions se vea claramente si el problema es de tests o de estilo/calidad. ESLint está configurado en `eslint.config.js` con las reglas de React hooks y react-refresh. Detecta cosas como hooks usados fuera de componentes, dependencias faltantes en `useEffect`, imports que no se usan, etc.

SAST con Semgrep

Semgrep analiza el código fuente **antes** de ejecutarlo, buscando patrones problemáticos. Lo corro en los 4 servicios incluyendo el frontend. El repo ya traía reglas custom en `backend/semgrep-rules/` que detectan:

- `hardcoded-secret.yaml` — variables con nombres tipo `password`, `token`, `apikey` que tengan un string literal asignado
- `no-eval.yaml` — usos de `eval()`, que básicamente es ejecutar código arbitrario
- `unvalidated-input.yaml` — inputs de `req.body` o `req.params` que se usan directo sin validar

Además de esas reglas propias, uso `--config=auto` que descarga las reglas oficiales de Semgrep para JavaScript/TypeScript. Si encuentra algo con severidad `ERROR`, el pipeline se corta.

Lo que no puede hacer Semgrep es detectar vulnerabilidades en runtime ni en dependencias externas. Para eso está Trivy.

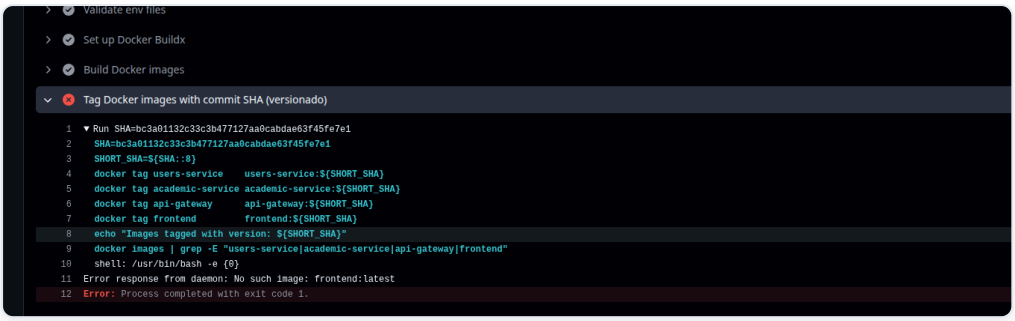
Docker build + versionado

Construyo las imágenes con `docker compose build` y después las etiqueto con el SHA corto del commit:

```
SHORT_SHA=${GITHUB_SHA::8}
docker tag users-service users-service:${SHORT_SHA}
```

Esto sirve para saber exactamente qué código hay dentro de cada imagen. Si después de un deploy aparece un bug, puedo identificar de qué commit viene la imagen que está corriendo y hacer rollback a la versión anterior. Sin versionado todas las imágenes se llaman `latest` y no hay manera de distinguirlas.

Durante el desarrollo del pipeline este step falló porque el servicio `frontend` en `docker-compose.yml` no tenía el campo `image:` definido, por lo que `docker tag` no encontraba la imagen. El error se ve acá:



SCA + seguridad de contenedor con Trivy

Trivy escanea las imágenes ya construidas buscando CVEs conocidos. Lo importante acá es que no solo mira las dependencias de Node — también escanea el sistema operativo base de la imagen. Si la imagen usa una versión de Alpine o Debian con alguna vulnerabilidad conocida, Trivy lo reporta.

Configuré `exit-code: 1` con `severity: HIGH,CRITICAL` en todos los servicios. Eso significa que si Trivy encuentra algo grave, el pipeline no avanza. No es solo informativo — actúa como gate real.

Genero los reportes en formato tabla y los subo como artefactos para tener evidencia descargable de cada run.

Smoke test

Al final levanto todo con `docker compose up -d`, espero 15 segundos y hago un `curl` al `/health` del api-gateway. Es una prueba mínima pero cubre el caso más básico: que los contenedores arranquen y se puedan comunicar entre sí. Si el api-gateway no responde, algo falló en el arranque o en la red interna de Docker.

El `if: always()` en el shutdown garantiza que los contenedores se bajen aunque algún step anterior haya fallado. Sin eso, un runner de GitHub Actions podría quedar con contenedores corriendo.

Por qué esto importa aunque el sistema "ya funcione"

Que la app funcione no significa que sea segura. Un sistema puede hacer exactamente lo que se espera y aun así tener un `const JWT_SECRET = "miclavesuper123"` hardcodeado en el código, o estar usando una versión de

`express` con una vulnerabilidad de path traversal, o tener una imagen base con 40 CVEs conocidos.

Sin el pipeline, esos problemas solo se descubren cuando alguien los explota. Con el pipeline, se detectan antes de llegar a producción, en cada commit, de forma automática.

El login en particular no se asume seguro por el hecho de funcionar. Cada vez que alguien toca `auth.controllers.js` o `auth.facades.js`, Semgrep lo analiza y Trivy verifica que las dependencias de `jsonwebtoken` y `bcrypt` no tengan CVEs reportados.

Resumen rápido

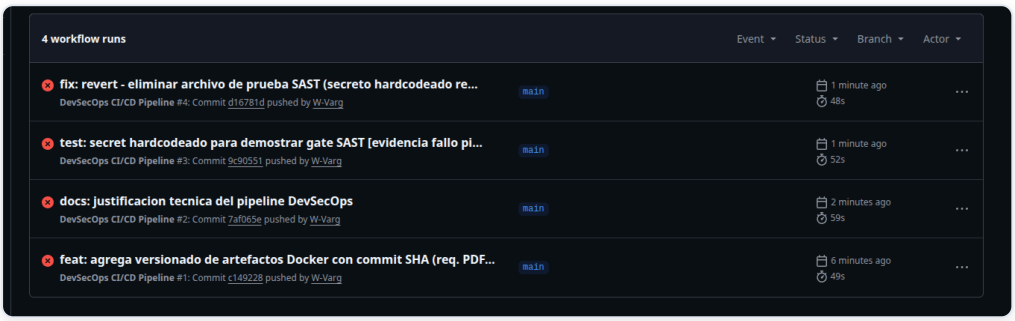
Herramienta	Para qué la uso	Cuándo falla el pipeline
`npm ci`	Instalación determinista	`package-lock.json` desincronizado
ESLint	Calidad y malas prácticas en frontend	Errores de lint
Jest	Tests unitarios de los 4 servicios	Cualquier test falla
Semgrep	SAST — busca patrones inseguros en el fuente	Hallazgo con severidad ERROR
Docker build	Construir y versionar las imágenes	Error de compilación/build
Trivy	CVEs en dependencias e imagen base	CVE HIGH o CRITICAL
Smoke test	Verificar que los contenedores arrancan	`/health` no responde

Pipeline completo en: `.github/workflows/devsecops.yml`
Ejecuciones en: https://github.com/W-Varg/Practica2/actions

Evidencia de ejecución

Run con errores

Ejemplo de un run fallido durante el desarrollo — el pipeline se cortó correctamente antes de llegar al build, cumpliendo el objetivo de "fail fast":



Run exitoso

Una vez corregidos todos los issues (CVEs, ESLint, imagen Docker sin tag), el pipeline pasa verde de punta a punta: