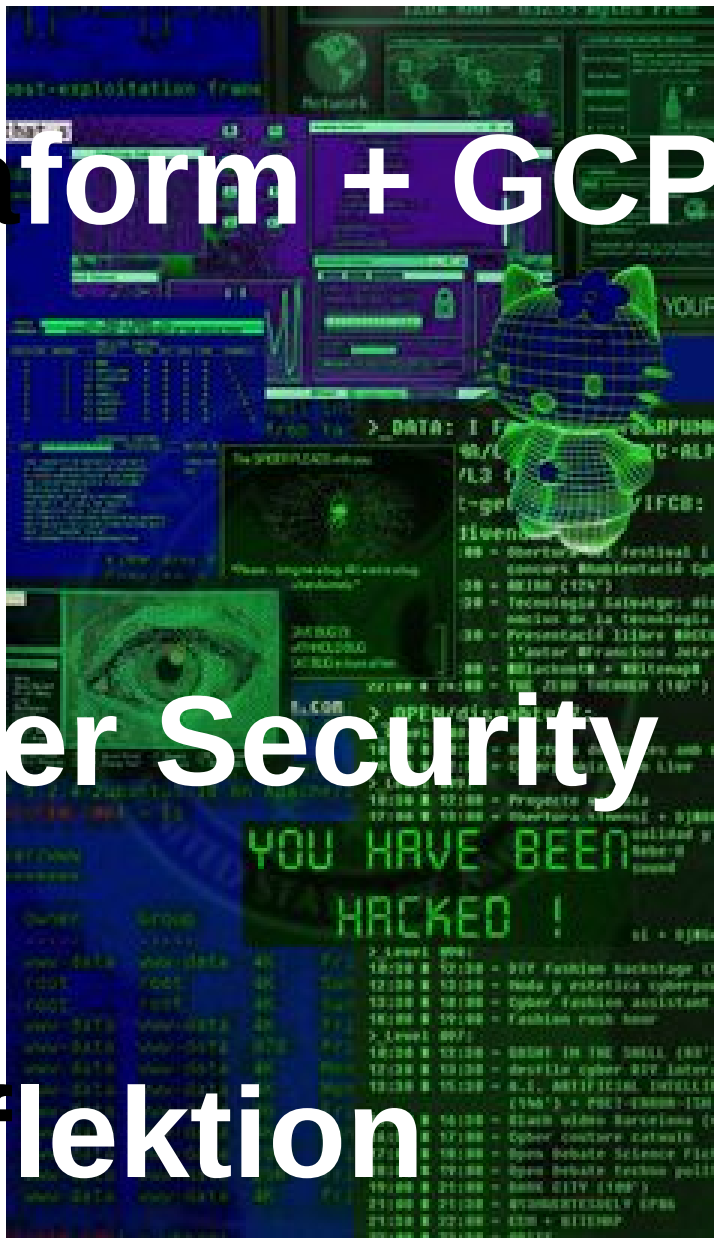
Individuell Rapport
Labb 1 och Labb 2
Daniel Wagenius
Chas Academy

Labb 1: Terraform + GCP



Labb 2: Container Security

Kursreflektion

Innehållsförteckning

Inledning

Mål med Labb 1

Labb 1 går ut på att automatisera driftsättningen av en virtuell Linux-server på Google Cloud med hjälp av Terraform. Genom GitHub Actions säkerställer man att ändringar testas innan de mergas och implementeras på servern.

Mål med Labb 2

Labb 2 går ut på att jämföra sårbarheten mellan en sårbar och en härdad Dockerfile och att se hur Policy-as-Code fungerar med hjälp av Gatekeeper.

Labb 1

Labbens flöde och utbildarens instruktioner

1. Skapa GitHub-repository, klona lokalt, och skapa filerna som ska användas:
   lab1-terraform/
   +-- main.tf
   +-- variables.tf
   +-- outputs.tf
   +-- terraform.tfvars (i .gitignore!)
   +-- .github/workflows/
   +-- terraform.yml
   Skapa en branch med namnet *features/initial-setup* och jobba mot den.
2. Redigera filerna och lägg till den tillhandahållna koden i filerna. Se även till att ändra användare och region i koden för att lyckas starta en virtuell maskin på rätt sätt.
   Se till att *.gitignore* har alla filer och mappar vi inte vill ha på GitHub.
3. Lägg till en backup-strategi i *main.tf.*
4. Lägg till en GitHub Actions Pipeline i *.github/workflows/terraform.yml*.
   Skaffa en GCP SA-nyckel för att kunna deploya till Google Cloud.
   Lägg till deploy-jobb i *terraform.yml*. Se till att inte pusha någon känslig data till GitHub.
5. Skapa en pull request från *features/initial-setup* till main. Vid merge bör en pipeline köras i GitHub Actions och den virtuella maskinen skapas och startas.
6. Skriv en *README.md* med information om labben.

Labbarkitektur

Labben använder Terraform för att köra en pipeline som syftar till att skapa och starta en virtuell maskin på Google Cloud Platform.

- *main.tf* är ritningen som beskriver vad du vill bygga. I vårt fall vill vi bygga en virtuell maskin på GCP. Den definierar den virtuella maskinen, nätverket och backup-regler.
- *variables.tf* talar om för Terraform vilka variabler som finns i projektet men inte vad dessa variabler har för värde.
- *terraform.tfvars* är komplementet till *variables.tf* och anger värdet på variablerna.
- *outputs.tf* skriver ut viss information, namnet på den virtuella maskinen, IP-adressen och vilken zon i nätverket den hamnat på, efter att Terraform har körts.
- *.github/workflows/terraform.yml* är instruktionerna för GitHub som talar om för GitHub att installera Terraform, köra *terraform init*, och sedan *terraform apply*.
- *startup.sh* innehåller instruktioner för att uppdatera systemet och installera säkerhetsmjukvara.

Labben använder GitHub för versionskontroll och för att automatiskt köra Terraform. Utvecklare pushar kod till en utvecklar-branch, och när denna branch mergas till main kommer GitHub att se *.github/workflows/terraform.yml* och utföra instruktionerna i Terraform-filerna.

GitHub Actions automatiserar kodgranskning med Lint, säkerhetsskanning med Trivy, och validering av konfigurationen med Terraform Validate.

Det är viktigt att utvecklare inte pushar nycklar, lösenord eller användarnamn till GitHub eller annan publik lagring online. Därför ska känsliga filer finnas i *.gitignore*, och nycklar och lösenord ska bara lagras i och hanteras av GitHub Secrets.

Implementation

Konfiguration

Den virtuella maskinen installeras med Ubuntu 22.04 LTS på en e2-micro i europe-north1-c.

Ubuntu 22.04 LTS (Long Term Support) är inte den senaste versionen av Ubuntu, men den är vältestad och uppdateras fortfarande. Den är därför stabil och säker, och verktygen vi använder i labben är garanterat kompatibla.

Vi installerar Ubuntu på en e2-micro-instans. Det är en väldigt liten maskin som erbjuds gratis till Google Cloud-användare. Den har tillgång till två vCPUs, men är begränsad till motsvarande 0,25 CPU-kärnor och har 1 GB RAM-minne. Det är begränsade resurser men duger gott till vår labb.

Från början var regionen satt till europe-north1-a. Googles gratis-servrar fylls upp väldigt fort, och det var rent tur att det fanns en ledig på europe-north1-c vid rätt tillfälle.

När den virtuella maskinen har skapats skickas *startup.sh* till GCP API, och när maskinen startas första gången ser en Google-tjänst skriptet och kör det automatiskt.
Vad skriptet gör:

- Uppdaterar systemet
- Installerar UFW och Fail2Ban
- Installerar unattended-upgrades, vilket verkar vara säkerhetspatchar
- Konfigurerar UFW att neka inkommande trafik, tillåta utgående trafik, och tillåta SSH
- Loggar att skriptet körts

Backup-strategi

I steg 3 lägger vi till en snapshot-policy i main.tf. Den säger till systemet att göra en backup varje dag klockan 3. Tiden är vald för att det ofta är lägre trafik som påverkas av att maskinresurser används till annat just då.

CI/CD Pipeline

Med *.github/workflows/terraform.yml* instrueras GitHub att köra tre jobb; Lint, Trivy och Validate. Jobben körs samtidigt. Om något jobb misslyckas så kommer hela pipelinen stanna och inte gå vidare till driftsättning.

Lint ser över formatering och syntax i koden. I min labb stoppade den allt för att den hittade en rad med ett mellanslag för lite. Lint ska även kunna se att en resurs man vill använda inte finns i regionen man har valt.

Trivy säkerhetsskannar vår pipeline. Den letar efter sårbarheter och osäkra konfigurationer. Ett exempel skulle kunna vara att någon har lagt en nyckel helt öppet i koden.

Validate kontrollerar att alla moduler är korrekt konfigurerade och till exempel att alla variabler är definierade. Den ser även till att koden är körbar.

Det är en del skillnader mellan hur Lint och Validate bedömer kod. Validate kan se att man har använt en måsvinge eller parentes fel, men inte att koden har rätt indragning. Lint å sin sida kan se att det är en felaktig indragning, men inte om en måsvinge eller parentes är fel. Validate bryr sig om att koden fungerar, Lint att koden är snyggt skriven och följer bra kodstandard.

Säkerhet och härdning

Härdning

Vi vill inte att den virtuella maskinen vi lagt upp i Google Cloud är helt öppen och sårbar mot internet. Därför utför vi enkla men viktiga säkerhetsåtgärder.

Skriptet *startup.sh* körs på den virtuella maskinen, och installerar UFW, Fail2ban och Unattended Upgrades.

UFW, eller Uncomplicated Firewall, är trots namnet inte en brandvägg utan ett lättanvänt gränssnitt för att hantera brandväggsregler. Genom skriptet stängs inkommande trafik av, utgående trafik tillåts, och SSH tillåts för att administratörer ska kunna ansluta till servern vid behov. UFW används för att det är enkelt och är standard på Ubuntu. Med användarvänligheten är det teoretiskt en mindre risk att man råkar låsa ute sig. Men det finns alltid människor som skulle klara det, så det är viktigt att administratörerna är kompetenta.

Fail2ban är ett verktyg för att stoppa intrångsförsök. Det övervakar inloggningsförsök och om en IP-adress misslyckas för många gånger på en kort tid stängs adressen av från att kommunicera med servern under en förutbestämd tid.

Unattended Upgrades är Ubuntus automatiserade säkerhetsuppdateringar. Det söker automatiskt och regelbundet efter säkerhetsuppdateringar och installerar uppdateringar som klassas som säkerhetskritiska. Det installerar inte hela systemuppdateringar, vilket är bra då det finns en risk att såna inte är kompatibla med de verktyg man använder. Det körs automatiskt i bakgrunden för att inte vara i vägen.

GitHub Secrets

Man vill i allmänhet inte att nycklar, lösenord och ofta användarnamn hamnar synligt på internet. Om det händer öppnar man upp sitt projekt för intrång av hotaktörer. Därför sparar man istället nycklar, lösenord och användarnamn som GitHub Secrets, ett krypterat lagringsutrymme. När koden sedan körs kommer det istället vara en variabel som GitHub letar efter i GitHub Secrets. Ingen utifrån kommer åt nyckeln, men den går fortfarande att använda i flödet för att autentisera sig.

Utförande

Vi fick tydligt skrivna instruktioner på hur labben skulle gå till och vad vi skulle göra, steg för steg. Vi fick även veta kraven för G och för VG. All kod tillhandahölls av utbildaren. Vi fick begära nycklar till Google Cloud. Det enda vi behövde göra var att lägga in våra egna användarnamn i koden och ställa in GitHub Secrets.

Det första jag gjorde var att skapa ett GitHub-repo och klona det. Jag skapade en utvecklar-branch som jag skulle jobba i, *features/initial-setup*. Därefter skapade jag alla filer som skulle användas. Respektive fil fick den tillhandahållna koden. Koden konfigurerar Terraform att starta en virtuell maskin. Det är givetvis alltid viktigt att *.gitignore* innehåller känsliga filer så att dessa inte hamnar på GitHub.

Efter steget där jag lade till koden till filerna kom steget där jag lade till backup-strategin i *main.tf*. Förstår inte helt varför den koden inte fanns med från början, men jag gissar att den är separat för att vi ska uppmärksamma.

Även *.github/workflows/terraform.yml* fick kod. Den filen aktiverar GitHub Actions och kör jobb där. Jobben är, som tidigare nämnts, Lint, Trivy och Validate.

Jag minns inte exakt men jag tror att det var här jag körde *terraform init* då alla filer som behövdes för det var färdiga. Det initierade arbetsmappen och laddade ner nödvändiga providers (drivrutiner) för att förbereda en anslutning till GCP.

För att driftsätta behövde jag en GCP SA-nyckel (Google Cloud Platform Service Account). Den begärde jag på utbildarens läroplattform Team Flags. Nyckel var en fil i JSON-format. Den lade jag till som en GitHub Secret, och hänvisade till den i *.github/workflows/terraform.yml*.

Till slut var allting färdigt för att pusha till GitHub, där jag kunde göra en pull request från *features/initial-setup* till *main*. När det gjordes kördes en pipeline automatiskt. Denna pipeline körde igenom jobben med ett godkänt resultat på första försöket. Jag lyckades tydligen väldigt bra med nyckelhantering och liknande.

Terraform-koden vi fått kör inte *terraform apply*, utan den körde jag manuellt när Terraform var initierat och jobben godkända. Det innebär att jag bara hade en CI-pipeline (Continuous Integration) och CD-delen (Continuous Deployment) gjordes manuellt.

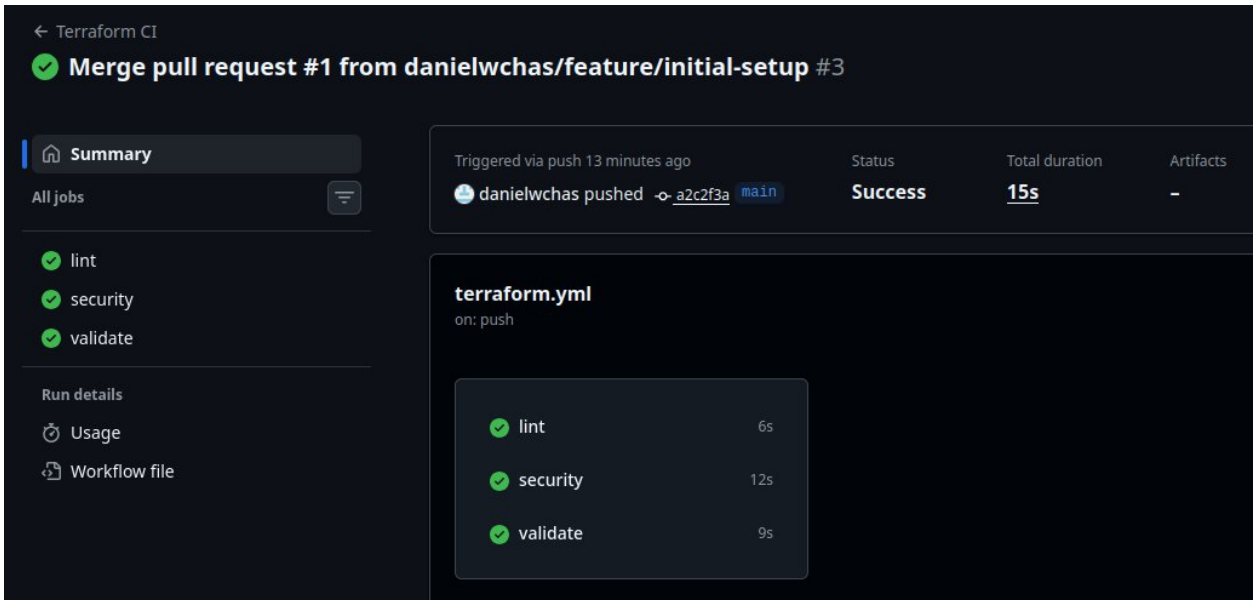
Det här var vad som krävdes för att få Godkänt på labben. Jag försökte mig inte på så mycket av VG-kraven. Jag hittade CIS Benchmarks för Ubuntu 22.04 och hur man blockerar CRITICAL i pipelinen. I koden vi fick upptäcks både CRITICAL och HIGH, men det finns ingenting som stoppar flödet om det händer. Då behöver man lägga till *--exit-code 1* i kodblocket.

De andra kraven bör inte vara alltför krångliga, jag fick bara ont om tid och ville inte stressa in något.

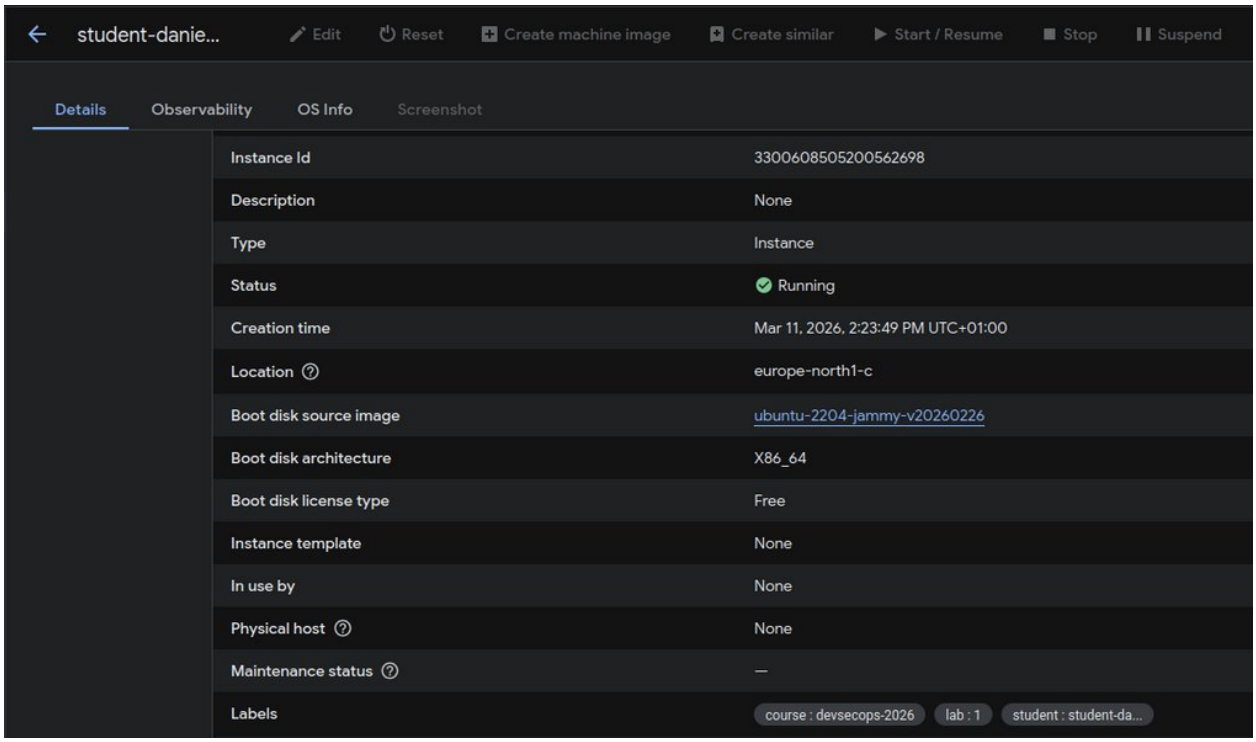
Resultat och validering

terraform init	Initierar arbetsmappen och laddar ner providers
terraform plan	Genererar en förhandsvisning av vad som kommer att skapas eller ändras
terraform apply	Tillämpar konfigurationen och driftsätter infrastrukturen i GCP

Pipelinen triggas genom att en pull request skapas från min feature-branch till main. Det säkerställer att ingen kod hamnar i main utan att godkännas i alla tester.



Terraform-pipeline efter PR



Den virtuella maskinen är skapad och aktiv

Labb 2

Labbens flöde och utbildarens instruktioner

1. Skanna en sårbar image. Skapa ett projekt med en sårbar app, installera Trivy, skanna projektet och spara resultatet av skanningen.
   lab2-container-security/
   +-- Dockerfile.vulnerable
   +-- Dockerfile.hardened
   +-- app.py
   +-- requirements.txt
   +-- sbom.json
   +-- scan-before.txt
   +-- scan-after.txt
   +-- require-labels-template.yaml
   +-- require-team-label.yaml
   +-- README.md
   +-- screenshots/
   +-- trivy-before.png
   +-- trivy-after.png
   +-- gatekeeper-block.png
2. Härda din Dockerfile.
3. Generera SBOM.
4. OPA Gatekeeper policy
5. Reflektion
6. Lämna in. Pusha filerna till GitHub och skriv en *README.md* med reflektioner.

Labbarkitektur

Labb 2 genomförs lokalt och på utbildarens läroplattform Team Flags. All kod tillhandahålls av utbildaren. Vi har begränsad tillgång till kluster med Gatekeeper, och därför fick vi använda Gatekeeper via Team Flags för att testa poddar.

- *app.py* är själva appen som ska byggas. I den här labben har den ingen mer funktion än att skriva ut en liten text.
- *Dockerfile.vulnerable* är den sårbara Dockerfilen.
- *Dockerfile.hardened* är den härdade Dockerfilen.
- *requirements.txt* säger till Docker att använda en nyare version av Flask.
- *sbom.json* är innehållsförteckningen på alla delar som använts

Utförande

Lokalt skapar vi till att börja med filstrukturen med en app och en Dockerfile. Vi lägger in koden i *app.py* och *Dockerfile.vulnerable*. Därefter bygger vi appen med Docker, för att sedan skanna containern med Trivy. Resultatet sparas i en textfil.

I nästa steg använder vi istället den härdade Dockerfilen, och *requirements.txt*. Vi lägger in den tillhandahållna koden, bygger och skannar containern. Även detta resultatet sparas i en textfil.

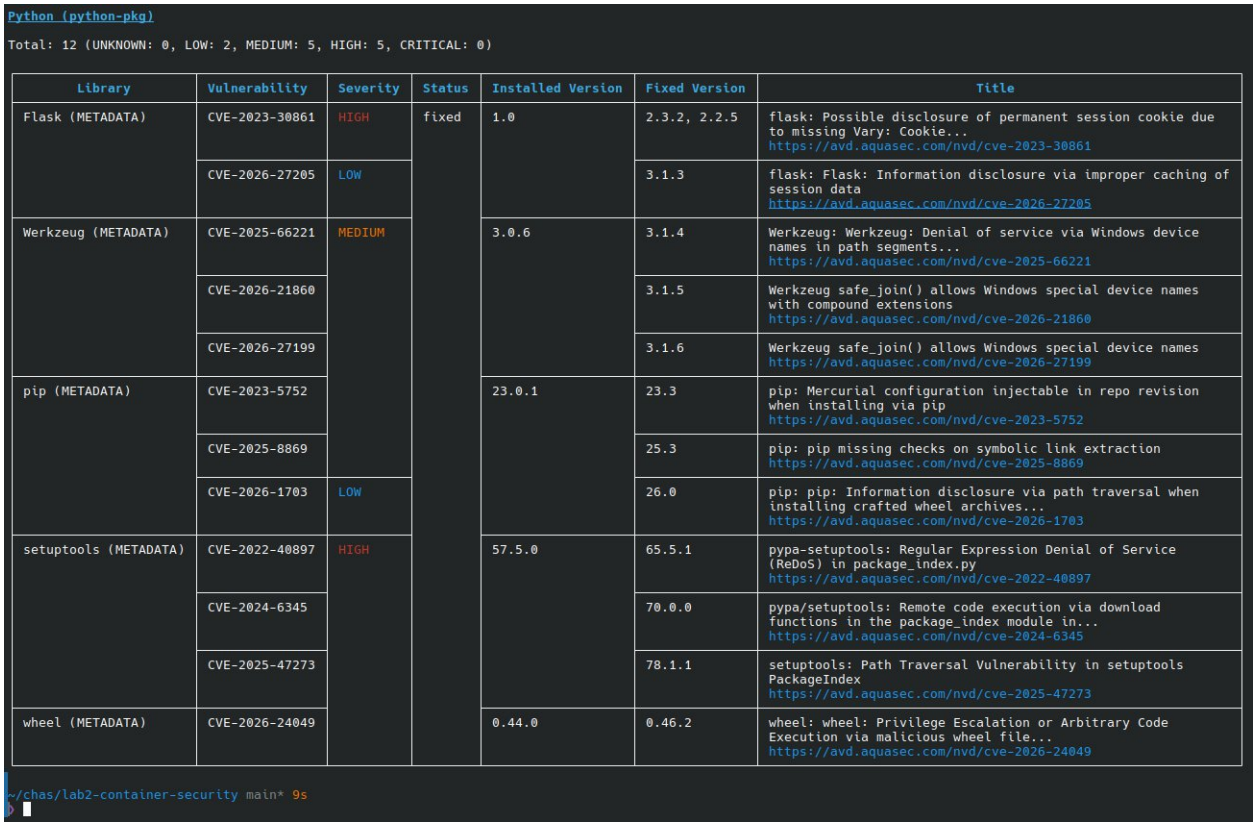
Det man ser direkt är storleken på containrarna. I den härdade appen används Python 3.12 Slim istället för den äldre och fullstora Python 3.8. Den härdade containern använder bara precis det den behöver och installerar aldrig bibliotek som inte används och som kan vara osäkra. Det minskar angreppsytan. Därför är den härdade containern betydligt mindre. Detsamma gäller resultatet av säkerhetsskanningarna. Den härdade containern har betydligt färre sårbarheter, och textfilen som loggar dessa är därför mindre än för den sårbara containern.

Nästa steg är att generera en SBOM (Software Bill Of Materials), en innehållsförteckning över vad som finns i projektet. Detta görs för att ha en översikt över vilka bibliotek och komponenter som används. Det är viktigt att hålla koll på ifall det upptäcks sårbarheter, eller för licensförändringar.

Det sista vi gör i labben är att testa poddar med Gatekeeper. Det görs på läroplattformen Team Flags. Det finns även instruktioner för att göra detta i terminalen, men på grund av rättigheter i klustret fungerade inte dessa. Tyvärr minns jag inte hur testerna gick till, och jag har bara skärmbilder kvar av resultatet. Men de visade sårbarheter där de skulle, och att den härdade podden var säkrare.

Resultat och validering

docker build -f Dockerfile.vulnerable -t my-app:vulnerable .
trivy image my-app:vulnerable
trivy image --severity CRITICAL,HIGH my-app:vulnerable > scan-before.txt



Python (python-pkg)

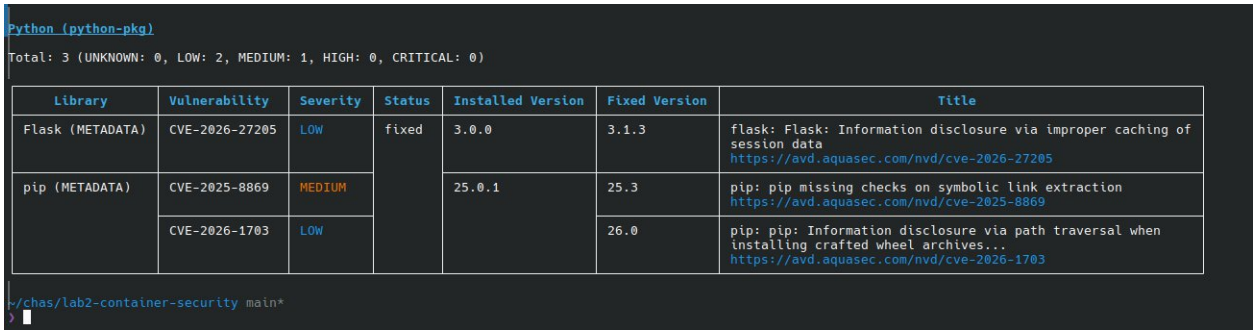
Total: 12 (UNKNOWN: 0, LOW: 2, MEDIUM: 5, HIGH: 5, CRITICAL: 0)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
Flask (METADATA)	CVE-2023-30861	HIGH	fixed	1.0	2.3.2, 2.2.5	flask: Possible disclosure of permanent session cookie due to missing Vary: Cookie... https://avd.aquasec.com/nvd/cve-2023-30861
	CVE-2026-27205	LOW			3.1.3	flask: Flask: Information disclosure via improper caching of session data https://avd.aquasec.com/nvd/cve-2026-27205
Werkzeug (METADATA)	CVE-2025-66221	MEDIUM		3.0.6	3.1.4	Werkzeug: Werkzeug: Denial of service via Windows device names in path segments... https://avd.aquasec.com/nvd/cve-2025-66221
	CVE-2026-21860				3.1.5	Werkzeug safe_join() allows Windows special device names with compound extensions https://avd.aquasec.com/nvd/cve-2026-21860
	CVE-2026-27199				3.1.6	Werkzeug safe_join() allows Windows special device names https://avd.aquasec.com/nvd/cve-2026-27199
pip (METADATA)	CVE-2023-5752			23.0.1	23.3	pip: Mercurial configuration injectable in repo revision when installing via pip https://avd.aquasec.com/nvd/cve-2023-5752
	CVE-2025-8869				25.3	pip: pip missing checks on symbolic link extraction https://avd.aquasec.com/nvd/cve-2025-8869
	CVE-2026-1703	LOW			26.0	pip: pip: Information disclosure via path traversal when installing crafted wheel archives... https://avd.aquasec.com/nvd/cve-2026-1703
setuptools (METADATA)	CVE-2022-40897	HIGH		57.5.0	65.5.1	pypa-setuptools: Regular Expression Denial of Service (ReDoS) in package_index.py https://avd.aquasec.com/nvd/cve-2022-40897
	CVE-2024-6345				70.0.0	pypa/setuptools: Remote code execution via download functions in the package_index module in... https://avd.aquasec.com/nvd/cve-2024-6345
	CVE-2025-47273				78.1.1	setuptools: Path Traversal Vulnerability in setuptools PackageIndex https://avd.aquasec.com/nvd/cve-2025-47273
wheel (METADATA)	CVE-2026-24049			0.44.0	0.46.2	wheel: wheel: Privilege Escalation or Arbitrary Code Execution via malicious wheel file... https://avd.aquasec.com/nvd/cve-2026-24049

~/chas/lab2-container-security main* 9s

Resultatet av Trivys säkerhetsskanning innan härdning

docker build -f Dockerfile.hardened -t my-app:hardened .
trivy image my-app:hardened
trivy image --severity CRITICAL,HIGH my-app:hardened > scan-after.txt



Python (python-pkg)

Total: 3 (UNKNOWN: 0, LOW: 2, MEDIUM: 1, HIGH: 0, CRITICAL: 0)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
Flask (METADATA)	CVE-2026-27205	LOW	fixed	3.0.0	3.1.3	flask: Flask: Information disclosure via improper caching of session data https://avd.aquasec.com/nvd/cve-2026-27205
pip (METADATA)	CVE-2025-8869	MEDIUM		25.0.1	25.3	pip: pip missing checks on symbolic link extraction https://avd.aquasec.com/nvd/cve-2025-8869
	CVE-2026-1703	LOW			26.0	pip: pip: Information disclosure via path traversal when installing crafted wheel archives... https://avd.aquasec.com/nvd/cve-2026-1703

~/chas/lab2-container-security main*

Resultatet av Trivys säkerhetsskanning efter härdning

Reflektioner

Termer och förkortningar som nämns i materialet och som jag kanske kommer använda i reflektionen
SRE: Site Reliability Engineering. Metodiken att bygga IT-infrastruktur på ett stabilt och driftsäkert, skalbart och automatiserat sätt.
Systemresiliens: Hur driftsäkert och motståndskraftigt mot störningar ett system är.
Automatisering: Att låta mjukvaran sköta det repetitiva och lite mer tråkiga, som tester och driftsättning. Minskar mänskliga fel och ökar effektiviteten.
IaC: Infrastructure as Code. Istället för att manuellt hantera och konfigurera IT-infrastruktur låter man kod göra det. Vi har använt Terraform för detta i labb 1.
Policy as Code: Säkerhetsregler skrivna som kod, som automatiskt valideras av systemet. I labb 2 använde vi Gatekeeper för detta.

Båda labbarna hörde ihop på flera sätt. Labb 1 lärde oss driftsätta en liten server på ett säkert sätt, med säkerhetstänk som till exempel GitHub Secrets och säkerhetsskanningar. Labb 2 visade på vikten av härdning av det man vill köra online och verktyg som upptäcker sårbarheter.

Flera delar av labbarna var automatiserade eller väldigt enkla med bra vertyg. I labb 1 skötte Terraform och GitHub actions sig själva efter att man pushade till GitHub och gjorde en pull request. I labb 2 byggde Docker containern själv och Gatekeeper kan ställas in att köra själv.

I tidigare kurser har vi fått leka runt lite med våra egna servrar, och vi har fått säkra upp dem med brandväggar och liknande. Men de flesta av oss har inte gjort nånting med servrarna utan bara lärt oss enklare inställningar och låtit servrarna köra utan nånting som körs mot internet på dem. I den här kursen har vi fått sätta upp en fungerande app på en server, och lärt oss hur vi ska göra det på ett säkert och smidigt sätt.

Jag har satt upp en server på Oracle Cloud där jag har driftsatt appen som min grupp använt i grupprojektet. Jag skaffade servern manuellt, och det kändes krångligt. Det dök upp många problem, speciellt på grund av att jag ville ha en gratis-server. När jag väl hade servern startade jag appen i en Docker-container på servern, och den är fortfarande igång där. Jag har inte lyckats skapa ett kluster på servern, och har inte ens tittat åt automatisering.

I labb 1 och grupprojektet fick vi lära oss hur man kan sätta igång en server på mycket smidigare sätt än jag gjorde med hjälp av IaC. Terraform sköter allting så länge man konfigurerar det rätt. Inget företag kommer vilja anlita någon för att köra en liten Docker-container på en lite gratis-server. Om det inträffar en driftstörning på min privata server så kommer min app inte längre fungera för den har ingen redundans. Om det inträffar en driftstörning på gruppens server så är Kubernetes-klustret självläkande och ser till att gruppens app fortfarande är tillgänglig.

Servern vi skapade i labb 1 har ingen redundans utan är bara en enda liten instans. Däremot är den lätt att återskapa om något skulle hända med den, tack vare Terraform.

Jag skulle kunna pusha nästan vilken osäker kod som helst till min privata server om jag inte är försiktig. På servern vi skapade i labb 1 kommer koden automatiskt testas (Lint och Trivy) och valideras (Terraform Validate), och de flesta sårbarheterna kommer upptäckas innan de ens kan hamna i main-branchen. Om man dessutom använder sig av Gatekeeper i klustret stoppas eventuella osäkra konfigurationer även om de skulle ta sig förbi pipelinen. Terraform ger även en bra bild av arkitekturen så att det är lättare att få en överblick av kostnader både i pengar och resurser.

Rollen som DevSecOps är förmodligen inte den jag vill ha helst av allt. Den är lite väl långt åt utvecklarhållet för min smak. Jag skulle hellre bli ren utvecklare i så fall. Men eftersom jag har hållit på med appar och servrar lite tidigare är det ändå intressant att lära sig hur processen fungerar om man gör på rätt sätt och inte laddar upp sitt huvudlösenord till ett publikt GitHub-repo av misstag när man är mitt i en onlinekurs. Och skulle jag bli erbjuden rollen skulle jag självklart ta den och förhoppningsvis göra bra ifrån mig.

Jag har inte tagit upp grupprojektet i den här rapporten då gruppen har skrivit en rapport specifikt för det projektet, men där har vi lärt oss hela kedjan från det enklare med Docker via Kubernetes till det mer avancerade och automatiserade Terraform. Jag hade förmånen att bli utsedd till gruppledare. Jag ser inte mig själv som en ledare, men gruppen har varit solid och alla har bidragit. När någon körde fast hjälptes vi åt och kom oftast vidare.

Den roligaste övningen i kursen var Red vs Blue i ”Operation Lockdown”. Operation Lockdown var en simulerad incidenthantering som utfördes i större grupper. Malmöstudenterna var Team Red och de som tillhörde Jönköping var Team Blue. Eftersom Jönköping är hemmaplan tillhörde jag Team Blue.Vår uppgift var att försvara det fiktiva företaget VaultCorp och härda deras system och täppa till läckor och sårbarheter.

Sett över hela kursen har vi fått känna på hur man jobbar inom DevSecOps. Vi har använt oss av CI/CD, hantering av känslig data, säkerhet och resiliens. Vi har även lärt oss incidenthantering i Operation Lockdown.

Länkar

Labb 1 Terraform
https://github.com/danielwchas/lab1-terraform

Labb 2 Container Security
https://github.com/danielwchas/lab2-container-security