Cloud security at Lyft
Agenda

- Overview: Lyft & our cloud environment
- Making cloud security happen at Lyft
  - Service abstraction
  - Resource orchestration
  - Identity & access controls
- Cloud-native security tactics
- Q&A
Overview: Lyft & our cloud environment
What is Lyft?

- Lyft is a rideshare service operating in the US and Canada
- Started as a hackathon project in 2012, the Lyft service has grown very rapidly: we now serve over one million rides/day
- From a tech standpoint:
  - Lyft is cloud-native—we have hosted our backend services in AWS since the first Lyft ride
  - Our engineering org is ~500 software engineers and many more tech users/consumers
  - We have a microservices architecture and operate ~400 services
Lyft’s engineering culture

• “Make it happen” is one of three core values at Lyft

• Engineers are empowered to—and accountable for—make it happen:
  – Devops model for service ownership, deployment, and maintenance
  – Heavy automation supporting SDL processes, CI/CD, monitoring, etc.
  – Few change management checkpoints with human gatekeepers
  – Making it happen: 200+ deploys/day

• Smaller central infrastructure & security teams, focused on automation, self-service, and supporting others who are (you guessed it) making it happen.

• YMMV
Lyft’s cloud (AWS) environment

- ~20k EC2 instances across 3 AZs
  - EC2 instances are single-tenant for Lyft applications, providing VM-level isolation of applications, data, credentials (more on this later)

- Use of many AWS products
  - Compute: EC2, Lambda
  - Networking: ELB, VPC, Route53, CloudFront
  - Data storage: S3, EBS, DynamoDB
  - Management: IAM, CloudTrail, CloudWatch

- Salt for cloud resource orchestration AND configuration management

- Service and orchestration changes deployed with Jenkins

- Microservice routing mesh with Envoy proxy server
Making cloud security happen at Lyft
Abstractions for isolation and trust

• At the scale of thousands of instances and millions of cloud resources, we need abstractions to help stay organized and reason about security policies

• Sounds kinda complicated, but we do this all the time:
  – Execution: server, virtual machine, namespace, process, etc.
  – Network: autonomous system, subnet, VLAN, security group, etc.
  – Directory service: groups, roles, etc.

• At Lyft we organize a number of the primitives AWS offers us into a rough abstraction we consider a service to help “make it happen”:
  – Single application deployed per service
  – Trust/access to resources inside service boundary
  – Default isolation from other services and resources outside service boundary
Defining a service at Lyft
Defining a service at Lyft

webservice-production-useast1
Defining a service at Lyft

webservice-production-useast1

SERVICE NAME | ENVIRONMENT | REGION
Defining a service at Lyft

Application web deployed from repo web

EC2 Instances tagged web-prod-useast1

account production, region us-east-1
Defining a service at Lyft

- Route53 CNAME: web-prod-useast1.lyft.net
- ELB: web-prod-useast1
- EC2 Autoscale Group: web-prod-useast1
- EC2 Security Group: web-prod-useast1
- EC2 Instances tagged: web-prod-useast1
- IAM Role/Instance Profile: web-prod-useast1
- DynamoDB Table: web-prod-useast1-users
- S3 Bucket Key: s3://backups/web-prod-useast
- EC2 Instances: web-prod-useast1
Defining a service at Lyft

**AWS**

- **EC2 Autoscale Group**
  - *web-prod-useast1*

- **Route53 CNAME**
  - *web-prod-useast1.lyft.net*

- **ELB**
  - *web-prod-useast1*

- **IAM Role/Instance Profile**
  - *web-prod-useast1*

- **EC2 Instances**
  - tagged *web-prod-useast1*

- **DynamoDB Table**
  - *web-prod-useast1-users*

- **EC2 Security Group**
  - *web-prod-useast1*

- **S3 Bucket Key**
  - *s3://backups/web-prod-useast*
Lessons learned: service definition

- Standardizing service and resource naming makes many things easier:
  - Default IAM policy maintains strong service boundary
  - Ownership, inventory, accounting
  - Creating a common mental model, making your docs higher-leverage
  - Templating and automation for service creation and maintenance
- Larger/complex services may need internal segmentation (or decomposition into smaller services) to achieve desired security properties
Cloud resource orchestration
Cloud resource orchestration

• Orchestration is essential to a secure, sanely-organized cloud
  – Known, managed workflow for making changes—no console or laptop changes
• Orchestration offers infrastructure-as-code, enabling
  – Code review & automated testing of infrastructure changes
  – Code repo as source of intent for analysis, incident response, etc.
• Enabling “make it happen”:
  – Service-specific resources are self-service and deployed with service repository
  – High-risk or account-/region-wide resources and default values are managed in a central repository
Self-service orchestration

Service Git repo

Service deploy

Service resource orchestration

Infrastructure Git repo

Infrastructure deploy

Account resource orchestration

Default/standard configuration
Templated self-service orchestration

- Lyft uses Saltstack, though Terraform and CloudFormation are now better choices
- Service templates are used to generate basic resource manifests for new services
- Resource names and policies based on service-specific variables (e.g. service name) allow creation of service-isolated sets of resources

https://github.com/lyft/confidant/
Templated self-service orchestration

Ensure {{ grains.cluster_name }} iam role exists:
  boto_iam_role.present:
    - name: {{ grains.cluster_name }}
    - policies:
      'iam':
        Version: '2012-10-17'
        Statement:
        - Action:
          - 'iam:ListRoles'
          - 'iam:GetRole'
          Effect: 'Allow'
          Resource: '*'
      'dynamodb':
        Version: '2012-10-17'
        Statement:
        - Action:
          - 'dynamodb:*
            Effect: 'Allow'
            Resource:
            - 'arn:aws:dynamodb:***:table/{{ grains.cluster_name }}'
            - 'arn:aws:dynamodb:***:table/{{ grains.cluster_name }}/***'

https://github.com/lyft/confidant/blob/master/salt/orchestration/confidant.sls
Templated self-service orchestration

Ensure {{ grains.cluster_name }} iam role exists:

```yaml
boto_iam_role.present:
  - name: {{ grains.cluster_name }}
  - policies:
    'iam':
      Version: '2012-10-17'
      Statement:
        - Action:
          - 'iam:ListRoles'
          - 'iam:GetRole'
          Effect: 'Allow'
          Resource: '*
    'dynamodb':
      Version: '2012-10-17'
      Statement:
        - Action:
          - 'dynamodb:*
          Effect: 'Allow'
          Resource:
            - 'arn:aws:dynamodb:*
            - 'arn:aws:dynamodb:*
            - 'arn:aws:dynamodb:*
```
Templated self-service orchestration

Ensure {{ grains.cluster_name }} iam role exists:

```python
def boto_iam_role_present:
  - name: {{ grains.cluster_name }}
  - policies:
      'iam':
        Version: '2012-10-17'
        Statement:
        - Action:
          - 'iam:ListRoles'
          - 'iam:GetRole'
          Effect: 'Allow'
          Resource: '*'
      'dynamodb':
        Version: '2012-10-17'
        Statement:
        - Action:
          - 'dynamodb:*'
          Effect: 'Allow'
          Resource:
            - 'arn:aws:dynamodb::*:table {{ grains.cluster_name }}'
            - 'arn:aws:dynamodb::*:table/{{ grains.cluster_name }}/*
```
Lessons learned: orchestration

- Source of truth in code repo makes self-improving infrastructure more difficult
- Fleet-wide changes (e.g. instance type upgrade) requires fleet-wide redeploy
- Fine-grained resource management is probably not the right level of abstraction for most teams
- Automated lint/static analysis to make sure orchestration changes are safe
- Orchestration deployment tools require high-privilege IAM role
  - Jenkins become high-risk large blast-radius infrastructure
  - How do you know your tests aren’t running with *:* IAM role?
Identity and access controls
Identity and access controls (for humans)

• AWS IAM has account-wide blast radius! Choose the strongest, best-managed tool you’ve got for managing IAM Users, Roles, and Policies.
  – For us: IAM Users + orchestration
  – For you? Could be SSO + IAM Roles. Consider whether to allow SSO to administrative roles.

• Enabling “make it happen”:
  – Self-service credential management: coinbase/self-service-iam
  – Allow engineers to list resources and elevate privileges for common ops tasks
  – Higher-risk and administrative access restricted and change-managed
IAM Users & Groups: “just enough” by default

GROUP: ENGINEERING

USER 1

- Action:
  - dynamodb:List*
  - ec2:Describe*
  - iam:Get*
  - iam:List*
  - s3:List*
  - ...

- Resource: *
- Effect: Allow

USER 2

- Action:
  - sts:AssumeRole

- Resource: admin-role
- Effect: Allow

GROUP: ADMINS
IAM Roles enable temporary elevated privilege

**GROUP: ADMIN**

**USER 1**

**USER 2**

**GROUP: ENGINEERING**

**ROLE: DEVOPS**

- Action: ec2:terminateInstances
- Resource: *
- Effect: Allow

**ROLE: ADMIN**

- Action: *
- Resource: *
- Effect: Allow
IAM Roles enable temporary elevated privilege

GROUP: ENGINEERING
- USER 1
- USER 2

GROUP: ADMINS

ROLE: DEVOPS
- Action: ec2:terminateInstances
- Resource: *
- Effect: Allow

ROLE: ADMIN
- Action: *
- Resource: *
- Effect: Allow
IAM policy to enforce MFA everywhere

GROUP: ADMIN

- USER 1
- USER 2

AssumeRole

GROUP: ENGINEERING

- USER 1
- USER 2

AssumeRole

ROLE: DEVOPS

Action: ec2:terminateInstances
Resource: *
Effect: Allow

ROLE: ADMIN

Action: *
Resource: *
Effect: Allow

aws:MultiFactorAuthPresent
IAM policy to enforce MFA everywhere

GROUP: ENGINEERING

USER 1

USER 2

GROUP: ADMINS

ROLE: DEVOPS

Action: ec2:terminateInstances
Resource: *
Effect: Allow

✔ aws:MultiFactorAuthPresent

ROLE: ADMIN

Action: *
Resource: *
Effect: Allow

✔ aws:MultiFactorAuthPresent

Locking down the AWS root user

- The root user of an AWS account cannot be constrained
  - Don’t use except when absolutely required (pen testing, billing changes, etc.)
- MFA is a must; we use physical tokens
- No credentials issued
- Alert on any use
Identity and access controls (for machines)

- Use IAM Roles everywhere—let AWS do the hard work to make this easy for you
  - Avoid the temptation to use IAM Users
  - Push partners to use roles with cross-account trust
- Protect the metadata service (http://169.254.169.254) when it matters:
  - Metadata Proxy for Docker containers: https://github.com/lyft/metadataproxy
  - SOCKS proxy for webhooks: https://github.com/stripe/smokescreen
Lessons learned: Identity and access controls

• IAM Users/Access Keys can quickly get messy AND have major consequences
  – Best case: critical production dependencies that are hard to change
  – Worst case: checked into source code/out on the Internet
  – Upshot: Use IAM Users only when you have no better alternative

• Have a plan for MFA enforcement and key rotation for all IAM Users

• Consider SSO for human users, at least for non-admin roles
  – Spend your time improving security, not resetting passwords
Cloud-native security tactics
Autoscaling → Autopatching

- Leverage the ephemeral nature of Instances to automate non-critical system patching
  - Requires system update on launch or continuously-updated AMIs/LaunchConfigurations

- Autoscaling as part of daily traffic load
  - Termination policy: OldestInstance or OldestLaunchConfiguration

- “Reaper Monkey”: explicitly terminating older instances
  - Blacklisting & scheduling to deal with more critical or stateful applications
Trust no one (else’s network)

- Cloud infrastructure ➔ isolated by default ➔ reduced blast radius
- Interconnecting office networks with cloud networks ➔ increased blast radius
  - Trust in administration of office network
  - Increased network scope for compliance assessment
- Consider running VPN terminator service inside your cloud network instead
  - Access from office = access from home = access from coffee shop
Brawn over brains

• AWS can sometimes make the easy things hard, but also makes hard things possible

• Using automation to leverage the incremental pricing and elastic nature of cloud resources can yield new solutions to old problems
  – AWS Lambda: massively parallel binary malware analysis: https://www.binaryalert.io/
  – AWS Organizations: create an AWS account per service/application for even greater isolation
  – AWS S3 + Athena: Collect all the data you want, and dig into it later only if you need to do incident response/etc.
Thank you

8 JUNE 2018