Week-4: Cybersecurity Architecture

This document has been prepared for the GWU SEAS-8405 Cybersecurity Architecture Course. This doctoral material critically examines prominent cybersecurity architectures and related software architectures. It delves into their core tenets, theoretical foundations, practical instantiations using contemporary cloud services (AWS/Azure), inherent complexities, known limitations, and relevant scientific literature.

Security Frameworks vs. Architecture

Understanding the distinction between **security frameworks** and **architectures** is crucial for effective risk management and secure system implementation. **Frameworks** provide comprehensive guidelines, standards, and best practices outlining *what* needs protection and governance. In contrast, **architectures** offer structured methodologies and principles that dictate *how* security solutions should be designed, organized, and deployed within technology infrastructures. The following table highlights key differences and examples to clarify their respective roles and contributions to cybersecurity.

Aspect	Framework	Architecture
Definition	A structured collection of guidelines, standards, and best practices for managing cybersecurity risks.	A systematic design and approach for implementing security solutions within technology systems.
Focus	Defines "what" should be protected and managed.	Describes "how" to structure and deploy security measures.
Examples	- NIST Cybersecurity Framework (CSF) - ISO/IEC 27001 - CIS Controls - Cloud Controls Matrix (CCM) - MITRE ATT&CK	- Zero Trust Architecture (ZTA) - Defense in Depth (DiD) - TOGAF Security Architecture - SABSA - Security by Design (SSDLC, DevSecOps)

Week-4: Cybersecurity Architecture

Chapter 1: Model-View-Control (MVC) Architecture

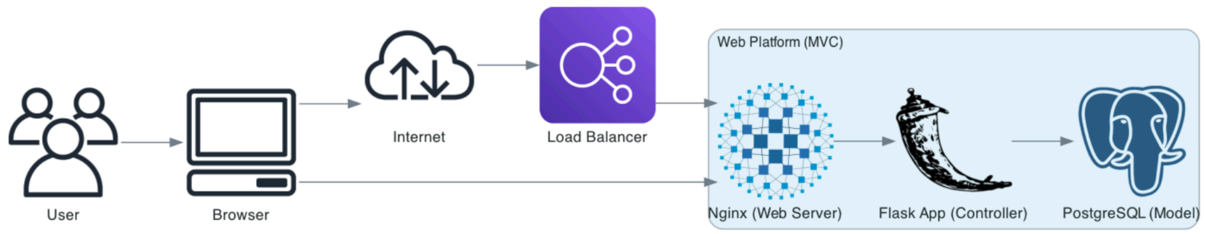
Core Principles

Model-View-Controller (MVC) is a software design pattern primarily used for developing user interfaces that divides an application into three interconnected components. While not strictly a *cybersecurity* architecture, its structure significantly impacts how security controls are implemented, particularly in web applications.

- **Separation of Concerns**:
    - **Model**: Manages the application's data, logic, and rules. Interacts with the database.
    - **View**: Represents the data to the user (the UI). Renders the Model data.
    - **Controller**: This role handles user input, interacts with the Model, and selects the View to render. It acts as the intermediary.
- **Modularity**: Components can be developed, tested, and maintained independently.
- **Reusability**: Components, especially the Model, can be reused.

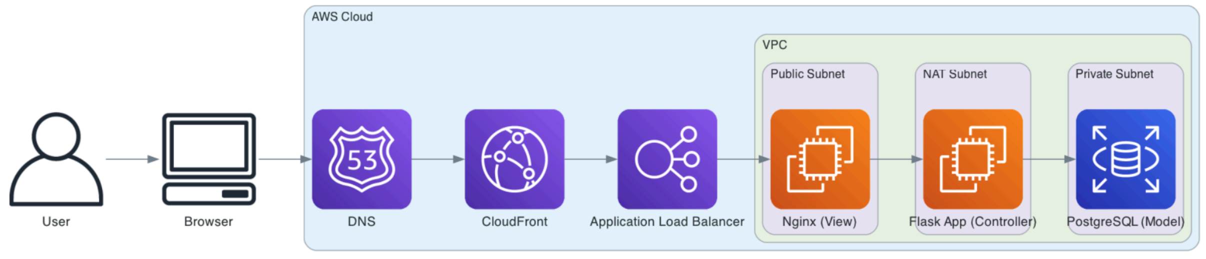
Web Platform with MVC

Throughout this document, we will develop a web platform in AWS and Azure using different architecture models.

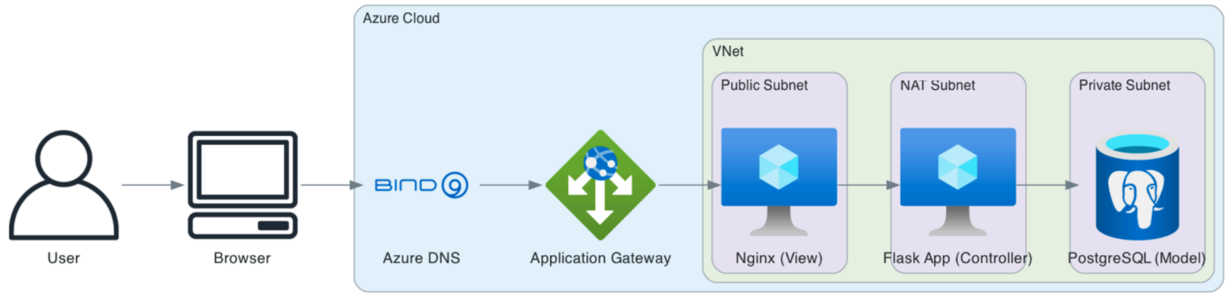


Flask Web Platform with MVC Architecture

Week-4: Cybersecurity Architecture



Flask Web Platform (MVC) on AWS



Flask Web Platform (MVC) on Azure

Use Cases & Platforms

- **When to Use**: Web applications (most common), graphical user interfaces (GUIs), applications requiring a clear separation between data, presentation, and logic for scalability and maintainability.
- **When Not to Use**: Straightforward applications or scripts where the overhead of separation is unnecessary. Tightly coupled systems are where separation is inherently tricky.
- **Appropriate Platforms**: Web application frameworks (Flask, Django, Ruby on Rails, ASP.NET MVC, Spring MVC), GUI toolkits.

Security Implications within MVC

Security must be considered in each component:

- **Model**:
  - **Data Validation**: Ensure data conforms to expected formats and rules before storage or processing. Protect against data tampering.

    - **Secure Database Interaction**: Use parameterized queries or ORMs to prevent SQL injection.
- **View**:
    - **Output Encoding/Escaping**: Correctly encode data before rendering it in HTML/JS/CSS to prevent cross-site scripting (XSS).
    - **Avoid Leaking Sensitive Data**: Ensure the View only displays data intended for the user, not sensitive internal state or excessive error details.
- **Controller**:
    - **Input Validation**: Validate all incoming data (parameters, headers, body) before passing it to the Model or View: check type, length, format, and range.
    - **Authorization Checks**: Before interacting with the model, ensure the user is authorized to perform the requested action. Enforce access control logic.
    - **Rate Limiting/Throttling**: Protect against denial-of-service and brute-force attacks.
    - **Secure Session Management**: Handle user sessions securely.

Issues & Considerations

- **Complexity**: Can introduce overhead for simple applications.
- **Learning Curve**: Teams need to understand the pattern to implement it correctly.
- **Well-Known Problems**:
    - **Fat Controller/Thin Model**: Business logic is incorrectly placed in the Controller instead of the Model, making it harder to test and reuse.
    - **Tight Coupling**: Poor implementation can lead to dependencies between components, negating the benefits of separation.
    - **Improper Validation**: Placing validation solely in the View (client-side) without server-side validation in the Controller or Model.
    - **Data Leaks**: Controller passing excessive or sensitive data from the Model to the View.
- **Known Cybersecurity Incidents**: Many web application breaches stem from failures within the MVC structure:
    - **SQL Injection**: Due to a lack of input validation in the Controller and a lack of parameterized queries in the Model's interaction with the database.

- **Cross-Site Scripting (XSS)**: Due to failure to properly encode output in the View.
- **Broken Access Control**: Due to missing or flawed authorization checks in the Controller.

References

- *Design Patterns: Elements of Reusable Object-Oriented Software* (Gang of Four) - Discusses related patterns.
- OWASP Secure Design Principles
- Documentation for specific MVC frameworks (e.g., Django, Rails).

Chapter 2: Defense in Depth (DiD)

Core Principles

Defense in Depth (DiD) is a security strategy employing multiple, independent security control layers to protect assets. If one layer fails, others are still in place to thwart an attack.

- **Multiple, Independent Layers**: Security controls span network, endpoint, application, and data domains.
- **Redundancy and Diversity**: Overlapping mechanisms prevent single points of failure.
- **Comprehensive Coverage**: Addresses diverse attack vectors (e.g., malware, insider threats).

Layered Controls

- **Network Layer**: This layer employs firewalls (stateful or next-generation), Intrusion Detection/Prevention Systems (IDS/IPS), and network segmentation (e.g., VLANs, subnets) to filter and monitor traffic.
- **Endpoint Layer**: This layer uses endpoint detection and Response (EDR), antivirus software, OS hardening (e.g., disabling unnecessary services), and timely patch management.

Week-4: Cybersecurity Architecture

- **Application Layer**: Implements Web Application Firewalls (WAF), secure coding practices (e.g., input sanitization), and rate limiting to thwart application-specific attacks.
- **Data Layer**: Leverages encryption (e.g., AES-256 for data at rest, TLS 1.3 for data in transit), Data Loss Prevention (DLP) tools, and access controls like Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC).
- **Monitoring and Recovery**: Integrates Security Information and Event Management (SIEM) for real-time log analysis, log aggregation, and disaster recovery mechanisms (e.g., backups, failover systems).

This approach assumes breaches are inevitable, emphasizing resilience over prevention alone.

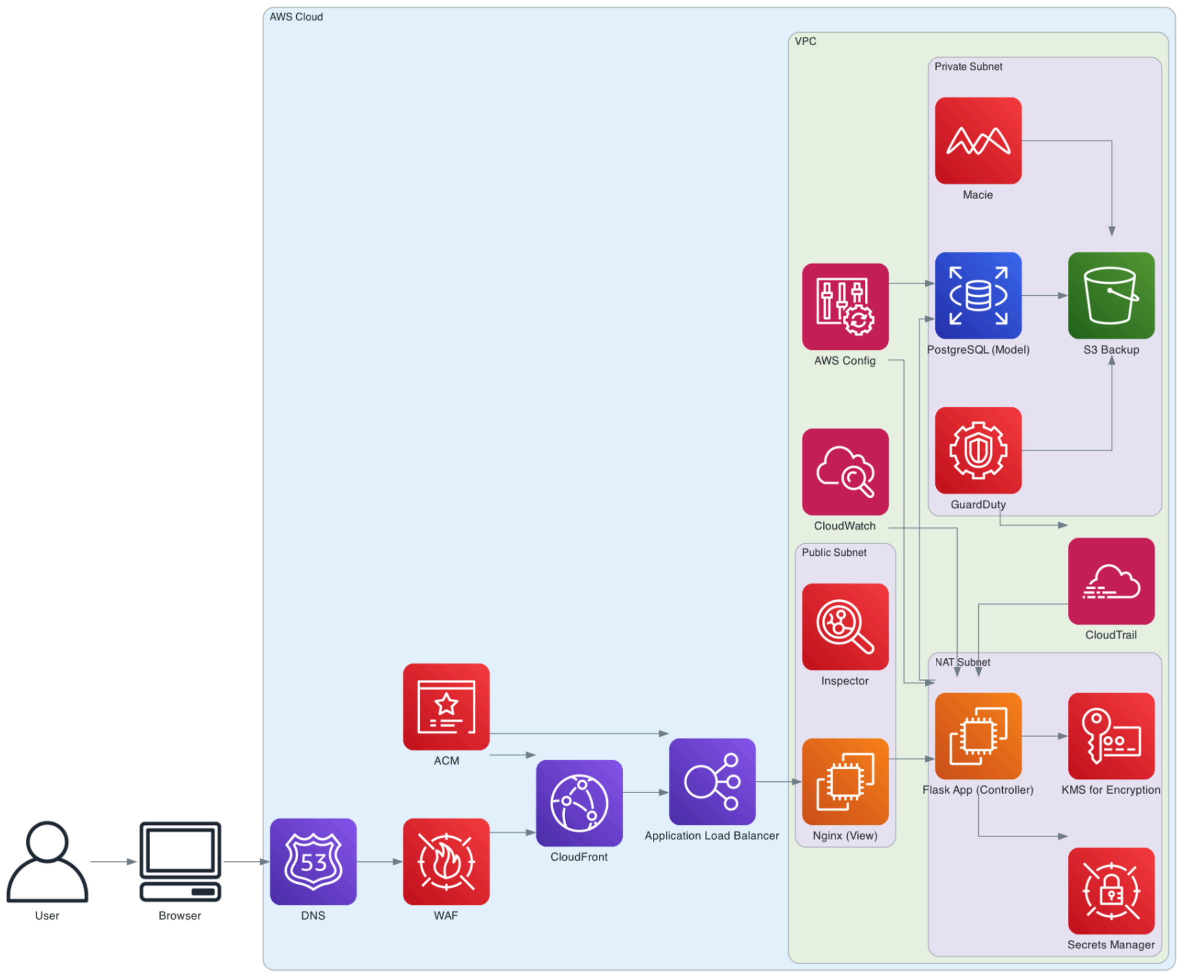
AWS & Azure Controls

DiD Layers	AWS Controls	Azure Controls	Reason & Functionality
User Access	IAM policies, MFA	Azure AD policies, MFA	Verify user identity, enforce least privilege access
Client Device	Browser security, TLS	Browser security, TLS	Prevent client-side exploits and ensure secure transport
Edge Protection	WAF, CloudFront, Route53, ACM	Azure Front Door, Azure CDN, Azure DNS, Azure Key Vault	Protect against DDoS, injection attacks, and ensure TLS
Network Perimeter	Security Groups, NACLs, ELB, VPC	NSGs, Azure Firewall, Azure Load Balancer, VNet	Define trusted boundaries and control ingress/egress
Public Subnet	Inspector, EC2 hardening	Azure Security Center, VM hardening	Scan for vulnerabilities and minimize the attack surface
App Layer (NAT Subnet)	Private EC2, Secrets Manager, KMS	Private VMs, Azure Key Vault	Secure app logic, encrypt data, manage secrets
Data Layer (Private Subnet)	RDS, S3 Backup, KMS	Azure SQL Database, Blob Storage Backup, Azure Key Vault	Encrypt and store sensitive data privately and durably
Monitoring & Logging	CloudWatch, CloudTrail	Azure Monitor, Azure Activity Log	Observe, alert, and audit operational behavior

Week-4: Cybersecurity Architecture

Compliance Monitoring	AWS Config	Azure Policy, Azure Security Center	Track and enforce compliance with best practices
Secrets Management	AWS Secrets Manager	Azure Key Vault	Store credentials securely and rotate them automatically
Threat Detection	GuardDuty, Inspector, Macie	Azure Security Center, Azure Defender, Azure Sentinel	Detect malware, sensitive data exposure, and API anomalies

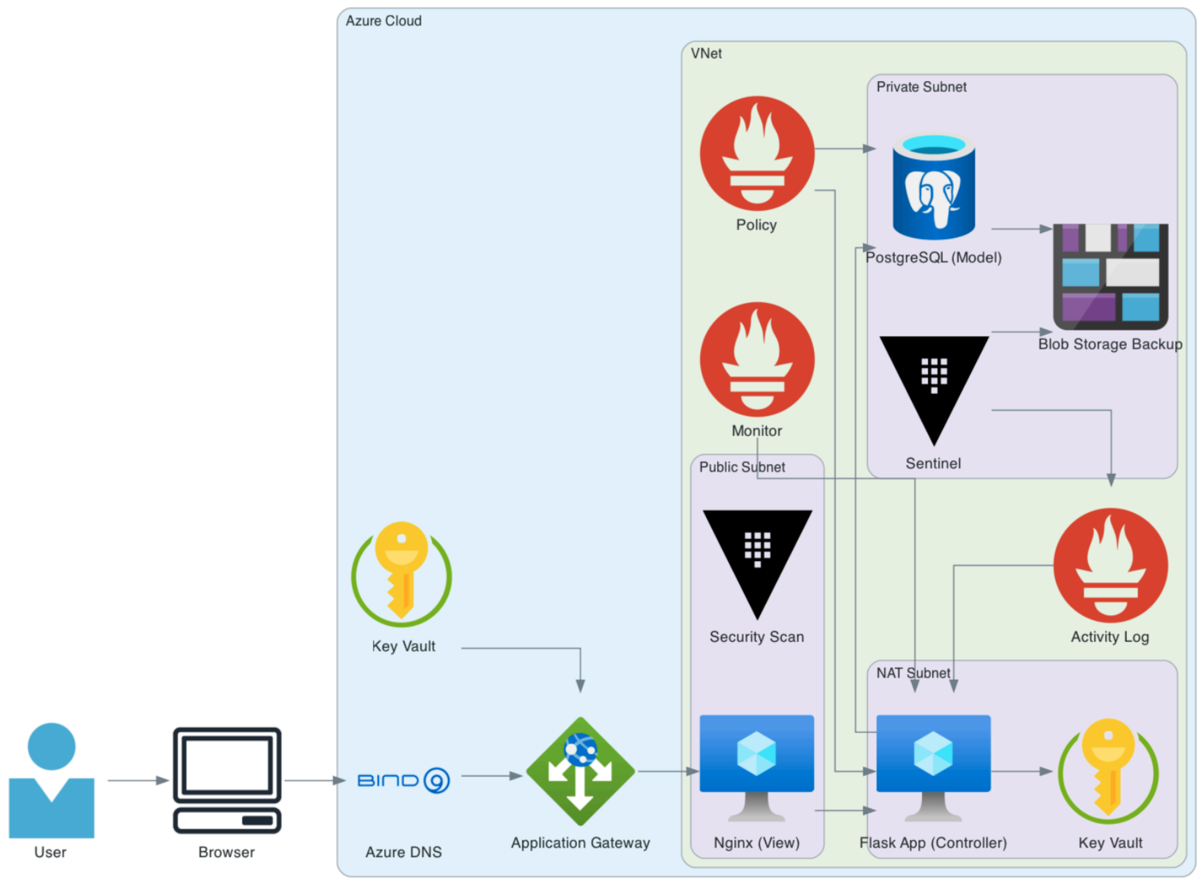
Web Platform with DiD in AWS



Flask Web Platform (MVC) on AWS - Defense in Depth

Week-4: Cybersecurity Architecture

Web Platform with DiD in Azure



Flask Web Platform (MVC) on Azure - Defense in Depth

Use Cases & Platforms

- **When to Use**: General-purpose enterprise security, protecting high-value assets and environments facing diverse threats (cloud, on-prem, hybrid). Essential for critical infrastructure.
- **When Not to Use**: May be overly complex or costly for very low-risk environments or resource-constrained scenarios. Environments needing minimal latency might find layers burdensome.
- **Appropriate Platforms**: Cloud (AWS, Azure, GCP), On-premises data centers, Hybrid environments.

Week-4: Cybersecurity Architecture

Issues & Considerations

- **Complexity**: Managing multiple layers and ensuring they work together cohesively can be challenging.
- **Cost**: Implementing and maintaining numerous security tools can be expensive.
- **Well-Known Problems**: Misconfigured firewalls or security groups, ineffective or stale monitoring, and over-reliance on perimeter defenses while neglecting internal layers.
- **Known Cybersecurity Incidents**: The Target breach (2013) exploited weak segmentation (a DiD principle) after initial compromise. The Equifax breach (2017) involved failure in patching (endpoint layer) and web application security (application layer).

References

- NIST. (2018). *SP 800-53: Security and Privacy Controls*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf
- CIS. (2023). *CIS Controls*. https://www.cisecurity.org/controls

Q&A with Example Scenarios

Scenario 1: Web Service Deployment (AWS)

- **Question**: How would DiD secure a public-facing web service?
- **Answer**: Deploy a WAF (e.g., AWS WAF) for application-layer filtering, security groups and NACLs for network isolation, EDR on EC2 instances, encrypt data with AWS KMS, and monitor with CloudWatch and GuardDuty.

Scenario 2: Kubernetes Cluster

- **Question**: How can DiD protect a Kubernetes-based application?
- **Answer**: Use network policies for pod segmentation, pod security standards, secrets management (e.g., HashiCorp Vault), container image scanning, and runtime monitoring with Falco.

Week-4: Cybersecurity Architecture

Multiple-Choice Questions

Based on a conceptual diagram showing layers: Network → Endpoint → Application → Data → Monitoring.

1. **What is the primary objective of DiD?**
    - a) Single-layer protection
    - b) **Layered redundancy and diversity**
    - c) Minimalist security
    - d) User authentication
2. **Which layer includes firewalls in DiD?**
    - a) **Network**
    - b) Endpoint
    - c) Application
    - d) Data
3. **What is a key benefit of redundancy in DiD?**
    - a) Reduced costs
    - b) **Prevention of single points of failure**
    - c) Simplified management
    - d) Faster deployment
4. **When is DiD least suitable?**
    - a) High-value assets
    - b) Cloud environments
    - c) **Low-risk, latency-sensitive systems**
    - d) Hybrid setups
5. **Which control is typical at the data layer?**
    - a) WAF
    - b) IDS
    - c) **Encryption**
    - d) Antivirus
6. **What is a common complexity issue in DiD?**
    - a) Lack of controls
    - b) **Overlapping configurations**
    - c) High performance

- d) Limited scalability

7. **What role does SIEM play in DiD?**
   - a) Data encryption
   - b) **Incident detection and response**
   - c) Network filtering
   - d) Endpoint hardening
8. **Which platform widely adopts DiD?**
   - a) Minimalist IoT devices
   - b) **Enterprise cloud systems**
   - c) Single-user apps
   - d) Legacy hardware
9. **What is a well-known DiD limitation?**
   - a) High speed
   - b) **Misconfigured controls**
   - c) Low cost
   - d) Simplicity

Chapter 3: Zero Trust Architecture (ZTA)

Core Principles

Zero Trust Architecture (ZTA) operates on the principle of "never trust, always verify." It assumes that threats exist both outside and inside the network, so no user or device is trusted by default, regardless of location. Verification is required for every access attempt.

- **Never Trust, Always Verify**: Continuously authenticate and authorize based on identity, device health, location, and other contextual data. Multi-factor authentication (MFA) is critical.
- **Least Privilege**: Grant users and systems only the minimum access required to perform their tasks.
- **Micro-segmentation**: Divide the network into small, isolated zones to limit lateral movement for attackers.

Week-4: Cybersecurity Architecture

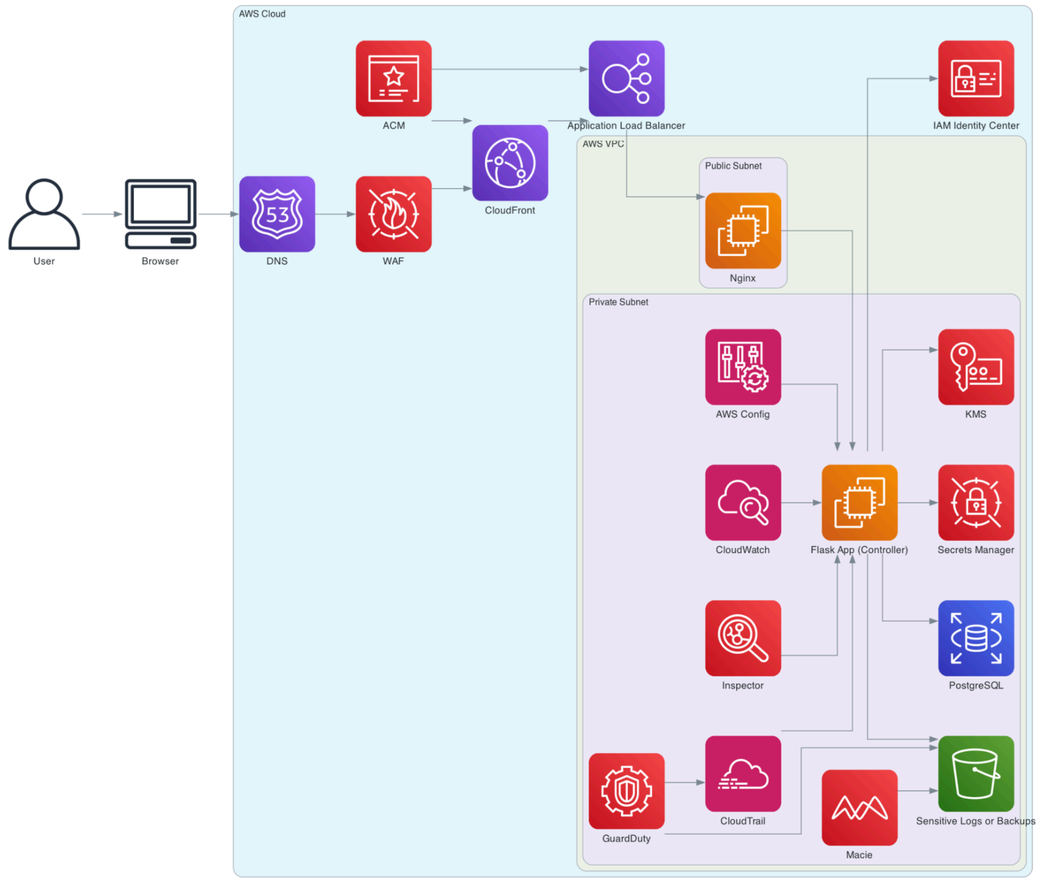
- **Encrypt Everywhere**: Encrypt data both at rest and in transit, including internal network traffic (e.g., using mTLS).
- **Continuous Authorization & Monitoring**: Access is not permanent. Re-verify access periodically or based on risk changes. Continuously monitor activity for anomalies.

AWS & Azure Controls

ZTA Pillar	**AWS Controls**	**Azure Controls**	**Purpose**
Identity Verification	IAM Identity Center	Azure AD	Ensure user authentication and least privilege access
Device Trust	Inspector	Azure Security Center / Microsoft Intune	Verify and assess compute instance/device security posture
Application Access	WAF, ELB, Secrets Manager	Azure Front Door, Azure Load Balancer, Azure Key Vault	Control access to the app and secure secret handling
Least Privilege	IAM policies and roles	Azure AD RBAC	Restrict access per job function and dynamic needs
Network Segmentation	VPC, Public/Private Subnets, Security Groups, NACLs	VNet, Subnets, Network Security Groups (NSGs), Azure Firewall	Limit lateral movement and isolate workloads
Encryption	KMS, ACM, Secrets Manager	Azure Key Vault	Encrypt data at rest and in transit
Continuous Monitoring	CloudWatch, CloudTrail, Config	Azure Monitor, Azure Activity Log, Azure Policy	Log activity and audit continuously
Threat Detection	GuardDuty, Macie, Inspector	Azure Defender, Azure Sentinel, Azure Security Center	Detect anomalous activity and data exposure
Data Access Control	S3 Policies, Secrets Manager	Blob Storage Policies, Azure Key Vault	Restrict and log sensitive data access
Data Exfiltration Protection	Macie, GuardDuty	Azure Sentinel, Azure Defender	Detect and block abnormal outbound data flows

Week-4: Cybersecurity Architecture

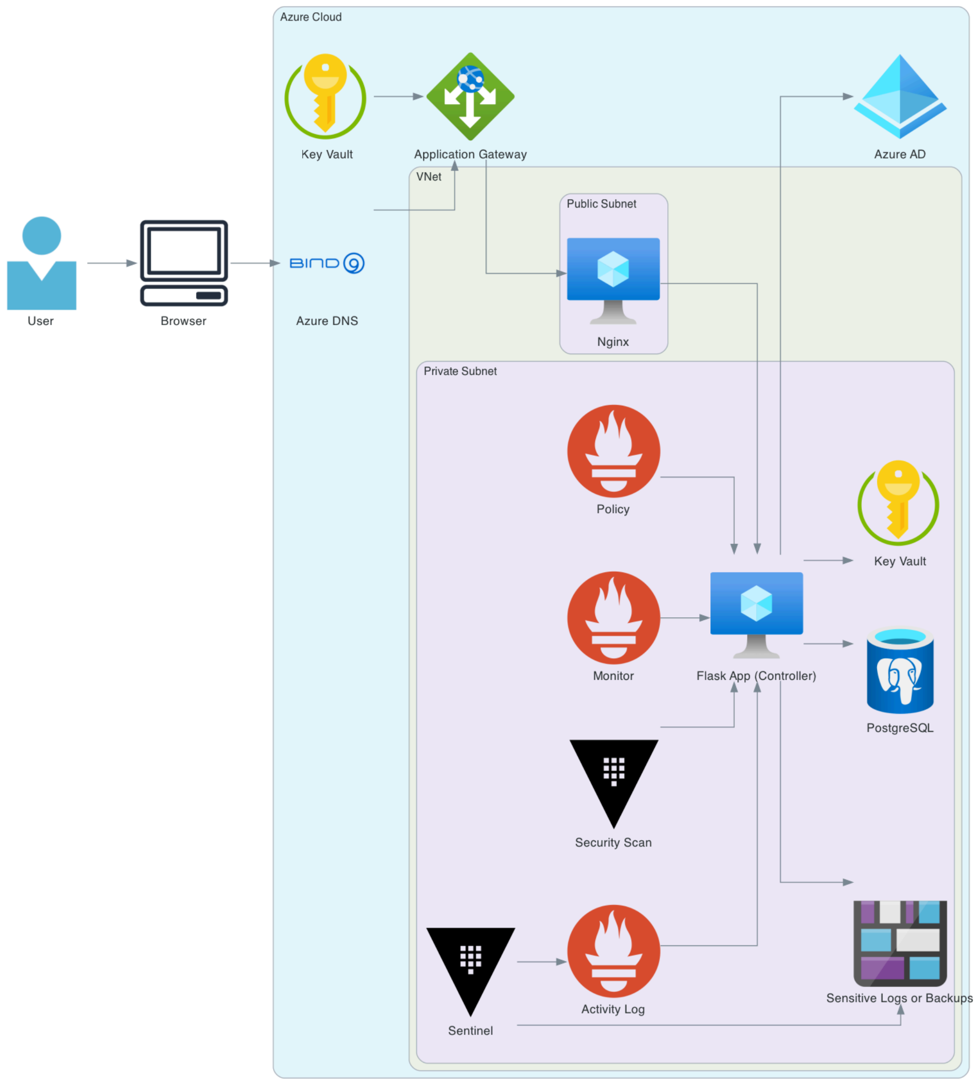
Web Platform with ZTA in AWS



Flask Web Platform on AWS - Zero Trust Architecture

Week-4: Cybersecurity Architecture

Web Platform with ZTA in Azure



Flask Web Platform on Azure - Zero Trust Architecture

Week-4: Cybersecurity Architecture

Use Cases & Platforms

- **When to Use**: Distributed environments, remote-first workforces, hybrid and multi-cloud setups, protecting sensitive data, and meeting compliance requirements.
- **When Not to Use**: May be overly complex for very small, static, low-risk environments. Legacy systems lacking modern identity and access management (IAM) capabilities can pose integration challenges.
- **Appropriate Platforms**: Cloud-native applications, SaaS platforms, environments leveraging modern IAM solutions (like Azure AD, Okta), containerized workloads.

Issues & Considerations

- **Complexity**: Implementing ZTA often requires significant changes to network architecture, identity systems, and workflows, which can make integration challenging.
- **User Experience**: Strict verification processes (like frequent MFA prompts) can sometimes impact user productivity if not implemented thoughtfully.
- **Well-known problems** include poorly managed token expiration policies, overreliance on identity proxies without sufficient device checks, and incomplete visibility across all assets.
- **Known Cybersecurity Incidents**: The SolarWinds breach highlighted supply chain risks that bypass traditional perimeters, reinforcing the need for internal ZTA controls. The Okta support system breach (2023) showed how compromised identity providers can undermine ZTA if not properly monitored and segmented. Colonial Pipeline (2021) involved compromised credentials, emphasizing the need for strong authentication and least privilege.

References

- NIST. (2020). *SP 800-207: Zero Trust Architecture*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf
- Google. (2018). *BeyondCorp Whitepaper*. https://cloud.google.com/beyondcorp

Week-4: Cybersecurity Architecture

Q&A with Example Scenarios

Scenario 1: Web Service Deployment

- **Question**: How does ZTA secure a web API?
- **Answer**: Use an identity-aware proxy (e.g., BeyondCorp), enforce MFA, apply least-privilege IAM roles, and segment API endpoints with network policies.

Scenario 2: Kubernetes Cluster

- **Question**: How can ZTA be implemented in Kubernetes?
- **Answer**: Deploy mTLS via a service mesh (e.g., Istio), use workload identity, enforce RBAC, and monitor with continuous telemetry.

Multiple-Choice Questions

Based on a diagram showing Identity → Device → Network → Application flow.

1. **What is ZTA's core principle?**
   - a) Perimeter defense
   - b) **Never trust, always verify**
   - c) Layered controls
   - d) Data encryption
2. **What component verifies user access in ZTA?**
   - a) Firewall
   - b) **IAM**
   - c) Antivirus
   - d) WAF
3. **What does micro-segmentation achieve?**
   - a) Faster networks
   - b) **Limits lateral movement**
   - c) Simplifies management

- d) Reduces costs

4. **When is ZTA most applicable?**
   - a) Legacy systems
   - b) **Cloud-native environments**
   - c) Low-risk apps
   - d) Static networks
5. **What is a key ZTA control?**
   - a) Static firewall rules
   - b) **Continuous authentication**
   - c) Endpoint hardening
   - d) Data backups
6. **What complexity arises in ZTA?**
   - a) Lack of tools
   - b) **Integration with legacy systems**
   - c) High performance
   - d) Simple deployment
7. **What role does device trust play?**
   - a) Encrypts data
   - b) **Ensures device compliance**
   - c) Filters traffic
   - d) Manages logs
8. **Which platform suits ZTA?**
   - a) **Hybrid cloud**
   - b) Single-user apps
   - c) Minimalist IoT
   - d) Legacy hardware
9. **What is a known ZTA issue?**
   - a) High speed
   - b) **User friction**
   - c) Low cost
   - d) Simplicity

Week-4: Cybersecurity Architecture

Chapter 4: Secure Software Development Lifecycle (SSDLC)

The Secure Software Development Lifecycle (SSDLC) integrates security practices into every phase of the software development process, from initial requirements gathering to deployment and maintenance. The goal is to build security in, rather than bolting it on afterward.

Core Principles

- **Security by Design**: Proactively design software with security in mind.
- **Shift Left**: Address security concerns as early as possible in the development cycle.
- **Continuous Testing**: Security testing (static, dynamic, dependency scanning) is regularly performed throughout the SDLC.
- **Risk Management**: Identify, assess, and mitigate security risks throughout development.

SSDLC phases include:

- **Requirements**: Conducts threat modeling (e.g., STRIDE, DREAD) and defines security specifications (e.g., authentication needs).
- **Design**: Performs secure architecture reviews and attack surface analysis to reduce exposure.
- **Coding**: Applies Static Application Security Testing (SAST) and adheres to secure coding standards (e.g., OWASP guidelines).
- **Testing**: Uses Dynamic Application Security Testing (DAST), fuzzing, and penetration testing to identify runtime issues.
- **Deployment**: Implements secrets management (e.g., HashiCorp Vault), signed artifacts, and Infrastructure as Code (IaC) validation.
- **Monitoring**: Integrates Runtime Application Self-Protection (RASP) and anomaly detection for post-deployment security.

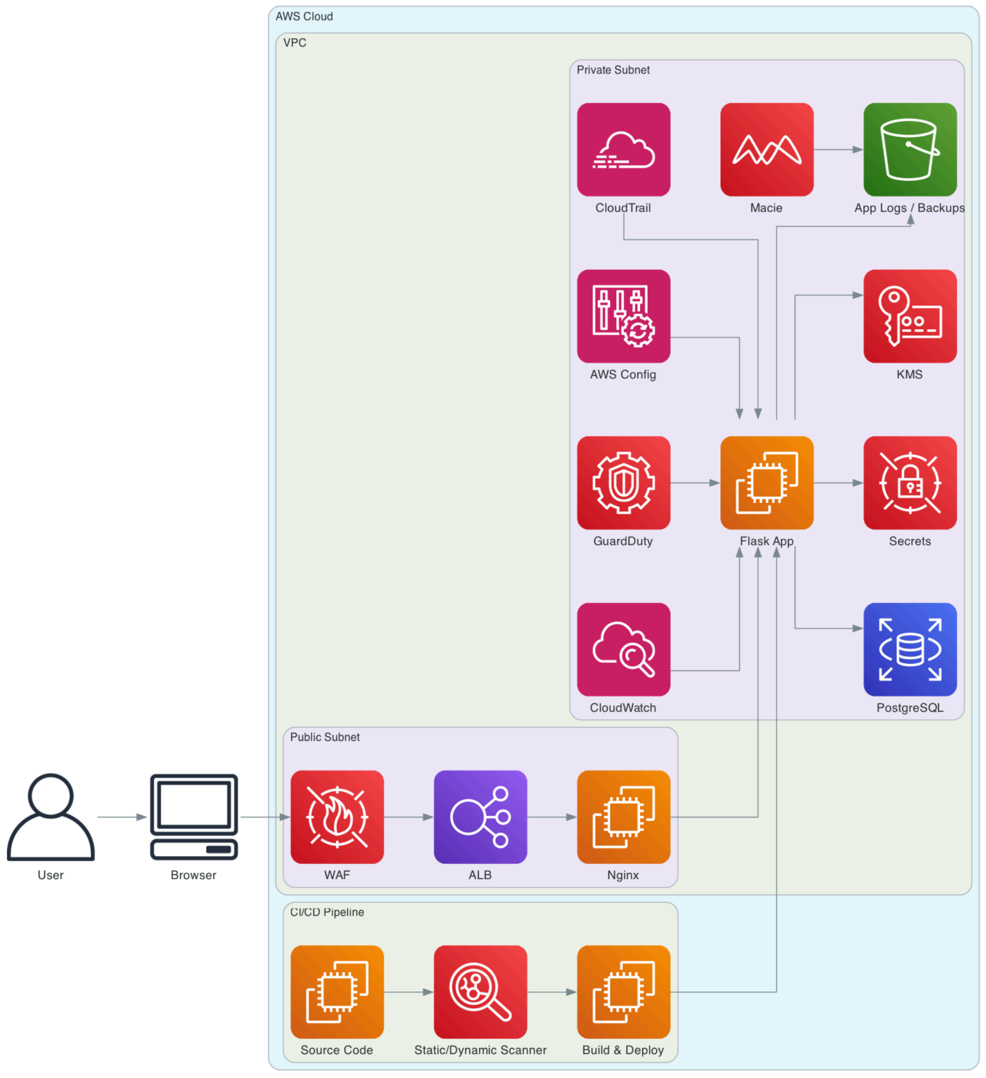
Week-4: Cybersecurity Architecture

AWS & Azure Controls

SSDLC Phase	AWS Controls	Azure Controls	Purpose
Secure Development	Source Code (CodeCommit), Static/Dynamic Scanner (CodeGuru, Partners)	Source Code (Azure Repos), Azure DevOps (SAST/DAST Extensions)	Identify vulnerabilities early in the code
Secure Build & Deploy	CI/CD Pipeline (CodePipeline, CodeBuild, CodeDeploy)	Azure Pipelines	Automated integration of security into releases
Perimeter Security	WAF, ALB (Public Subnet)	Azure Front Door, Azure Load Balancer	Control access and filter malicious traffic
Application Security	Nginx (Public), Flask App (Private), Secrets Manager (Private)	Nginx (Public), App Service (Private), Azure Key Vault	Secure runtime behavior and secret handling
Data Security	RDS (Private), S3 (Private), KMS	Azure SQL Database (Private), Blob Storage (Private), Azure Key Vault	Encrypt and protect structured/unstructured data
Monitoring & Threat Detection	CloudWatch, GuardDuty, Macie, CloudTrail, AWS Config	Azure Monitor, Azure Defender, Azure Sentinel, Azure Activity Log, Azure Policy	Continuously monitor and detect anomalies

Week-4: Cybersecurity Architecture

Web Platform with SSDLC in AWS



Flask Web Platform on AWS - Security by Design (SSDLC / DevSecOps)

Week-4: Cybersecurity Architecture

Use Cases & Platforms

- **When to Use**: Any organization developing software, especially applications handling sensitive data or operating in high-risk environments (finance, healthcare, government). Essential for DevOps and CI/CD pipelines.
- **When Not to Use**: Arguably, it is always beneficial, but the *intensity* might be scaled back for very small, internal, low-risk projects or one-off scripts. However, basic practices like dependency scanning are almost universally applicable.
- **Appropriate Platforms**: CI/CD platforms (Jenkins, GitLab CI, GitHub Actions, Azure DevOps), Development environments, and Cloud deployment platforms.

Issues & Considerations

- **Cultural Shift**: Requires developers, security teams, and operations to collaborate closely (DevSecOps). Needs buy-in and training.
- **Time & Resources**: Integrating security tools and processes can add time to development cycles and require investment in tools and expertise.
- **Well-Known Problems**: Skipping threat modeling, inadequate dependency hygiene (using outdated or vulnerable libraries), resistance from developers seeing security as a blocker, difficulty keeping pace in rapid release cycles.
- **Known Cybersecurity Incidents**: The SolarWinds attack involved malicious code being injected into the build pipeline, highlighting the need for build integrity checks (a core SSDLC principle). The Heartbleed vulnerability (2014) in OpenSSL demonstrated the impact of coding flaws in widely used libraries, emphasizing the need for secure coding and testing. Persistent vulnerabilities in Adobe Flash were partly due to failures in embedding security throughout its development.

References

- OWASP. (2021). *SAMM*. https://owaspsamm.org
- NIST. (2022). *SSDF*. https://csrc.nist.gov/publications/detail/sp/800-218/final

Week-4: Cybersecurity Architecture

Q&A with Example Scenarios

Scenario 1: Web Service Deployment

- **Question**: How does SSDLC secure a web app pipeline?
- **Answer**: Perform threat modeling, use SAST/DAST in CI/CD, scan dependencies, and deploy with signed artifacts.

Scenario 2: Kubernetes Cluster

- **Question**: How can SSDLC secure a containerized app?
- **Answer**: Validate container images, enforce secure configurations, and monitor runtime behavior.

Multiple-Choice Questions

Based on a diagram showing SSDLC phases: Requirements → Design → Coding → Testing → Deployment.

1. **What is SSDLC's primary goal?**
   - a) Rapid development
   - b) **Security by design**
   - c) Cost reduction
   - d) User authentication
2. **In which phase is threat modeling performed?**
   - a) **Requirements**
   - b) Coding
   - c) Testing
   - d) Deployment
3. **When is SSDLC most effective?**
   - a) One-off scripts
   - b) **Sensitive software projects**
   - c) Low-risk apps
   - d) Legacy code

Week-4: Cybersecurity Architecture

4. **What tool is used in the coding phase?**
   - a) Penetration testing
   - b) **SAST**
   - c) Firewall
   - d) SIEM
5. **What is a major SSDLC challenge?**
   - a) High speed
   - b) **Cultural resistance**
   - c) Low complexity
   - d) Minimal testing
6. **In SolarWinds, what SSDLC step was neglected?**
   - a) Requirements
   - b) **Build security**
   - c) Monitoring
   - d) Design
7. **What does dependency scanning address?**
   - a) Code efficiency
   - b) **Vulnerable libraries**
   - c) Network security
   - d) User access
8. **Which platform suits SSDLC?**
   - a) **CI/CD pipelines**
   - b) Static websites
   - c) Single-user apps
   - d) Hardware devices
9. **What is a common SSDLC issue?**
   - a) High performance
   - b) **Time delays**
   - c) Low cost
   - d) Simplicity

Week-4: Cybersecurity Architecture

Chapter 5: Zero Knowledge Architecture (ZKA)

Core Principles

Zero Knowledge Architecture (ZKA) is a system design approach where the service provider has mathematically verifiable zero access to the user's unencrypted data. Even during storage, processing, or transmission facilitated by the service, the provider cannot decrypt the user's content. The user retains sole control over their keys and data.

Goal: Ensure service providers cannot access user data, enhancing privacy and security.

Layer	Control Description
Encryption at rest	Use client-side encryption (CSE); providers store only ciphertext
Encryption in transit	TLS/SSL for all data exchanges (e.g., HTTPS, mTLS)
Key Management	End-user-controlled keys; e.g., HSM-backed BYOK or client-side KMS
Access Control	No privileged backdoor access for admins; zero-access policies
Authentication	Passwordless auth (SRP) or multi-party auth with no stored secrets on server
Auditing	Tamper-proof logging of every access attempt to encrypted data containers
Secure Sharing	End-to-end encrypted sharing links or proxy re-encryption
Zero Knowledge Proofs (ZKPs)	Use zk-SNARKs/ZKPs where the user can prove ownership/rights without revealing data
Local computation	Client-side rendering, indexing, and processing (e.g., E2EE messaging apps)

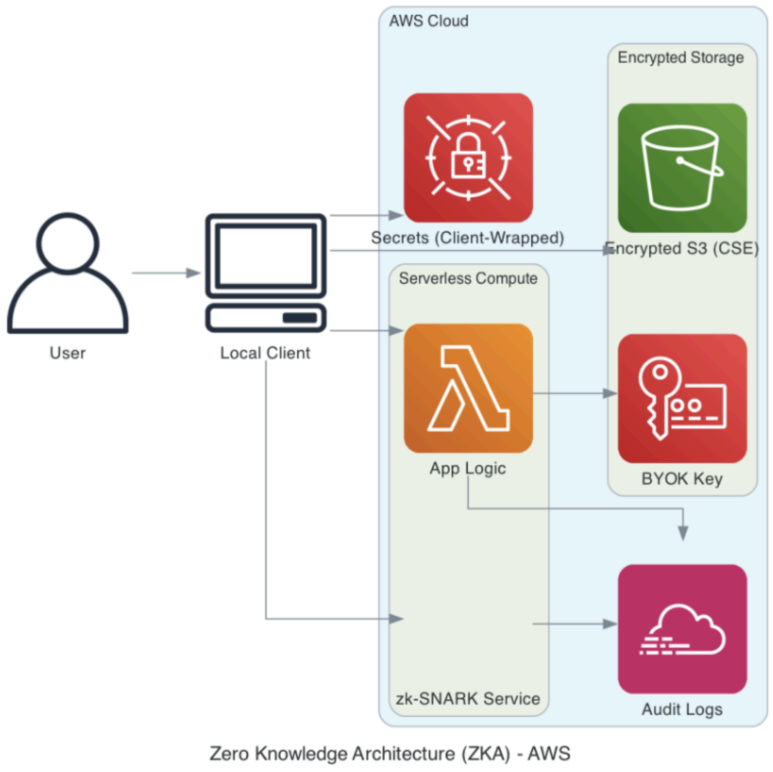
AWS & Azure Controls

ZKA Control	AWS Controls	Azure Controls	Purpose
Client-Side Encryption	CSE with KMS, S3 (Private)	Azure Blob with client-side encryption, BYOK	Ensure cloud providers never access plaintext data
Zero-Access Storage	S3 with denied admin access policies	Azure Blob + RBAC + no-ops access keys	Prevent backend/control plane access to data

Week-4: Cybersecurity Architecture

Key Ownership	AWS KMS (external), CloudHSM	Azure Key Vault with customer-managed keys (HSM-backed)	User controls cryptographic keys
End-to-End Encryption	E2EE in custom apps using AWS Lambda, Cognito	Azure Functions with E2EE libraries	Only clients encrypt/decrypt, not services
Zero Knowledge Proofs	zk-SNARK-enabled APIs (e.g., ZK-ID via partners)	Integration with zk-compatible apps (via custom code/VMs)	Prove identity or access without revealing data
Auditing	CloudTrail + object-level logging (S3 Access Logs)	Azure Activity Log + Storage Analytics	Track data access attempts without exposing contents

Web Platform with ZKA in AWS



Week-4: Cybersecurity Architecture

Issues & Considerations

- **Limited Functionality**: Server-side features like searching encrypted content, content analysis, or data recovery are difficult or impossible.
- **Key Management Complexity**: Users are responsible for their keys. Key loss means permanent data loss. Secure key recovery mechanisms are challenging to design without compromising the zero-knowledge principle.
- **Integration Challenges**: Integrating ZKA systems with third-party services that expect plaintext data access is difficult. Compatibility with AI/ML models running server-side is often limited.
- **Well-Known Problems**: Key loss = data loss, difficulty performing server-side operations, usability trade-offs.
- **Known Cybersecurity Incidents**: While the core data remains protected, metadata exposure has been a concern (e.g., who communicated with whom, even if the content is encrypted, as seen in warrants served to providers like ProtonMail). Confiscated encrypted devices often remain inaccessible to authorities, demonstrating the strength of user-controlled encryption.

References

- ProtonMail. (2021). *Security Whitepaper*. https://proton.me/security
- Signal. (2023). *Technical Documentation*. https://signal.org/docs

Q&A with Example Scenarios

Scenario 1: Web Service Deployment

- **Question**: How does ZKA secure a web-based file storage service?
- **Answer**: Encrypt files client-side with user keys, store ciphertext on servers, and use ZKPs for access verification.

Scenario 2: Kubernetes Cluster

- **Question**: How can ZKA protect a Kubernetes app?
- **Answer**: Encrypt pod data client-side, manage keys externally, and ensure no plaintext in cluster storage.

Week-4: Cybersecurity Architecture

Multiple-Choice Questions

Based on a diagram showing Client → Encryption → Server (Ciphertext Only).

1. **What is ZKA's primary goal?**
    - a) Server efficiency
    - b) **Zero provider access**
    - c) Network speed
    - d) User authentication
2. **Where does encryption occur in ZKA?**
    - a) Server
    - b) **Client**
    - c) Network
    - d) Database
3. **What ensures privacy in ZKA?**
    - a) Firewalls
    - b) **E2EE**
    - c) IDS
    - d) Monitoring
4. **When is ZKA most suitable?**
    - a) Server-side analytics
    - b) **Privacy-first apps**
    - c) Low-risk systems
    - d) Legacy apps
5. **What is a key ZKA component?**
    - a) WAF
    - b) **User-controlled keys**
    - c) SIEM
    - d) Antivirus
6. **What is a major ZKA challenge?**
    - a) High speed
    - b) **Key management**

- c) Low cost
- d) Simplicity

7. **What limits ZKA functionality?**
   - a) Encryption speed
   - b) **Server-side processing**
   - c) Network bandwidth
   - d) User access
8. **Which platform suits ZKA?**
   - a) **Encrypted messaging**
   - b) Public websites
   - c) IoT devices
   - d) Legacy systems
9. **What is a known ZKA problem?**
   - a) High performance
   - b) **Data loss from key loss**
   - c) Low complexity
   - d) Easy integration

Chapter 6: Adaptive Security Architecture (ASA)

Core Principles

Adaptive Security Architecture (ASA) is a framework that moves beyond static defenses to a model of continuous monitoring and response. It uses real-time telemetry, contextual analysis, and behavioral analytics to dynamically adjust security controls based on the evolving threat landscape and internal system behavior.

Goal: Use continuous context and behavioral analysis to *dynamically* adjust defenses in real-time, moving from incident response to continuous response.

Week-4: Cybersecurity Architecture

Layer	Control Description
Continuous Monitoring	Telemetry from endpoints, network, identities (SIEM/SOAR input)
Behavioral Analytics	UEBA (User & Entity Behavior Analytics) for anomaly detection
Context-Aware Access	Risk-based MFA, device posture check, geo-aware policies
Dynamic Policy Engine	Policies that adjust based on context and past behavior
Deception Tech	Honeypots or decoy environments for active threat engagement
Automated Response	Use SOAR to auto-quarantine, block IPs, alert, or require re-auth
Threat Intelligence	Feed external and internal intel into risk scoring and policies
Runtime Protection	RASP, WAF, or eBPF-based defense adjusted by threat levels
Feedback Loops	Learn from attacks and improve rules (e.g., ML + human triage)

AWS & Azure Controls

ASA Control	AWS Controls	Azure Controls	Purpose
Telemetry Collection	CloudWatch, GuardDuty, CloudTrail, VPC Flow Logs	Azure Monitor, Log Analytics, Defender, Network Watcher	Collect data across the stack for analysis
Behavioral Analytics	Detect anomalies with Macie/Lookout for Metrics/GuardDuty	UEBA with Microsoft Defender for Cloud, Sentinel UEBA	Identify abnormal behavior patterns
Context-Aware Access	IAM + device posture (AWS SSO Conditional Access)	Conditional Access Policies, Intune device compliance	Adjust access dynamically based on context (risk, device)
Dynamic Policy Enforcement	Lambda-based firewall/WAF rules, Config Rules Remediation	Azure Policy + Logic Apps/Azure Functions for enforcement	React in real time to policy violations or risk changes

Automated Response	SOAR via EventBridge + Lambda/Step Functions	Sentinel Playbooks (Logic Apps), Azure Automation	Contain threats without manual intervention
Threat Intelligence	AWS TI feeds (GuardDuty), GuardDuty custom threat lists	Microsoft Threat Intelligence Center (integrated in Sentinel/Defender)	Integrate external and internal intel for smarter defense

Use Cases & Platforms

- **When to Use**: Environments under active or sophisticated threat, large enterprises with complex attack surfaces, regulated industries requiring high visibility and rapid response capabilities, and environments needing real-time risk assessment.
- **When Not to Use**: Very small organizations with limited resources or expertise to manage complex monitoring and response systems. Low-risk, static applications with minimal real-time interaction needs.
- **Appropriate Platforms**: Security Information and Event Management (SIEM), Security Orchestration, Automation, and Response (SOAR), User and Entity Behavior Analytics (UEBA), Endpoint Detection and Response (EDR), Network Detection and Response (NDR), platforms incorporating Machine Learning (ML) for security analytics.

Issues & Considerations

- **Complexity**: Integrating diverse telemetry sources and tuning analytics engines requires significant expertise. Managing automation workflows (SOAR playbooks) is complex.
- **False Positives & Alert Fatigue**: Behavioral analytics can generate noise, leading to ignored alerts if not properly tuned. Automation based on false positives can disrupt operations.
- **Privacy Trade-offs**: Continuous monitoring, especially UEBA, can raise privacy concerns if not implemented transparently and ethically.
- **Well-known problems include** SIEM "alert fatigue," delays in updating dynamic policies, gaps in telemetry coverage, and difficulty integrating intelligence feeds effectively.

Week-4: Cybersecurity Architecture

- **Known Cybersecurity Incidents**: The Capital One breach (2019) reportedly involved SIEM alerts related to the WAF misconfiguration that were not acted upon quickly enough, highlighting the need for effective response (manual or automated) linked to monitoring. The Equifax breach involved delays in patching despite available threat intelligence, showing a failure in adapting defenses based on known risks.

References

- Gartner. (2017). *Adaptive Security Architecture*. https://www.gartner.com
- MITRE. (2023). *D3FEND Framework*. https://d3fend.mitre.org

Q&A with Example Scenarios

Scenario 1: Web Service Deployment

- **Question**: How does ASA secure a web app?
- **Answer**: Use SIEM for telemetry, UEBA for anomaly detection, and SOAR to auto-block malicious IPs.

Scenario 2: Kubernetes Cluster

- **Question**: How can ASA protect a Kubernetes cluster?
- **Answer**: Monitor with Falco, analyze pod behavior with UEBA, and auto-scale security policies.

Multiple-Choice Questions

Based on a diagram showing Telemetry → Analytics → Response flow.

1. **What is ASA's core principle?**
   - a) Static defense
   - b) **Real-time adaptation**
   - c) Data encryption
   - d) User authentication
2. **What drives ASA decisions?**

   - a) Firewalls
   - b) **Telemetry**
   - c) IDS
   - d) WAF
3. **What does UEBA detect?**
   - a) Network speed
   - b) **Behavioral anomalies**
   - c) Code errors
   - d) Hardware issues
4. **When is ASA most effective?**
   - a) Static apps
   - b) **High-risk environments**
   - c) Low-budget systems
   - d) Legacy setups
5. **What is a key ASA capability?**
   - a) Manual response
   - b) **Automated reaction**
   - c) Static rules
   - d) Data backups
6. **What is a major ASA issue?**
   - a) High speed
   - b) **False positives**
   - c) Low cost
   - d) Simplicity
7. **In Capital One, what failed ASA?**
   - a) Telemetry
   - b) **Response to alerts**
   - c) Encryption
   - d) Network security
8. **What role does SOAR play in ASA?**
   - a) Data encryption
   - b) **Automated response**
   - c) User access

    - d) Network filtering

9. **Which platform suits ASA?**
    - a) **SIEM/SOAR**
    - b) Minimalist apps
    - c) Single-user systems
    - d) Legacy hardware
10. **What is a known ASA problem?**
    - a) High performance
    - b) **Alert fatigue**
    - c) Low complexity
    - d) Easy integration

Chapter 7: SolarWinds SUNBURST Supply Chain Attack

What Went Wrong in the SolarWinds Hack?

The SolarWinds cyberattack, discovered in December 2020, was a sophisticated, state-sponsored (attributed to Russia's SVR) supply chain attack targeting the SolarWinds Orion Platform, a widely used IT infrastructure management software. The core elements of the attack involved:

1. **Initial Access & Reconnaissance:** Attackers gained access to SolarWinds' internal network months before the main attack, likely through password spraying or exploiting vulnerabilities. Weak credentials (e.g., `solarwinds123` reportedly found on a public GitHub repository) may have facilitated this.
2. **Build System Compromise:** The attackers infiltrated SolarWinds' software development and build environment. They meticulously studied the build process for the Orion platform.
3. **Code Injection (SUNBURST):** The attackers subtly injected malicious code (a backdoor later named SUNBURST) into the source code of a legitimate Orion component (`SolarWinds.Orion.Core.BusinessLayer.dll`). This was done directly within

the build pipeline, likely by compromising a build server or modifying source code before compilation.

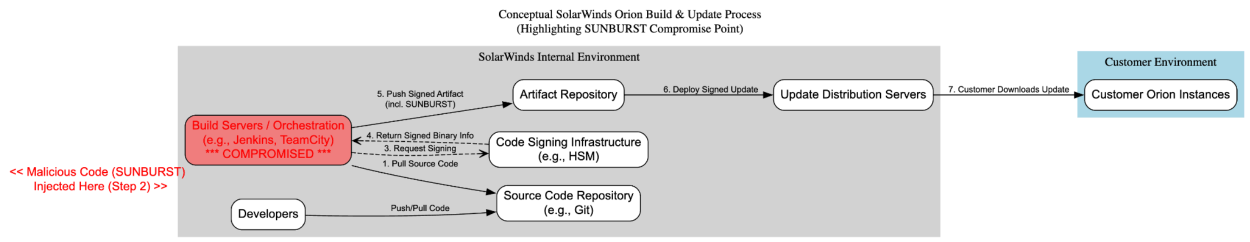
4. **Malicious Code Signing:** Crucially, the compromised build process compiled the legitimate SolarWinds code *along with* the SUNBURST backdoor. The resulting malicious DLL was then signed with a valid SolarWinds digital certificate, making it appear authentic and trustworthy.
5. **Distribution via Legitimate Updates:** The trojanized DLL was distributed to potentially tens of thousands of SolarWinds Orion customers worldwide through the platform's standard software update mechanism.
6. **Dormancy & Evasion:** Once installed in customer environments via the update, the SUNBURST backdoor remained dormant for approximately two weeks. It performed checks to ensure it wasn't running in analysis environments (sandboxes) and used sophisticated techniques to blend its command-and-control (C2) traffic with legitimate Orion activity (e.g., using HTTP requests to domains designed to mimic legitimate traffic patterns).
7. **Targeted Second-Stage Payload:** After the dormancy period and initial C2 communication, the attackers selectively deployed second-stage payloads (like TEARDROP and RAINDROP) onto high-value targets identified among the compromised organizations (including numerous US government agencies and Fortune 500 companies).
8. **Lateral Movement & Data Exfiltration:** Using the second-stage payloads, the attackers moved laterally within victim networks, escalated privileges (often targeting identity systems like Active Directory Federation Services - ADFS), created persistent backdoors, and exfiltrated sensitive data (e.g., emails).
9. **Detection Failure:** The attack went undetected for many months due to its stealth, the use of signed binaries, sophisticated C2 techniques, and likely gaps in monitoring and threat detection capabilities both at SolarWinds and within victim organizations. It was ultimately discovered by the cybersecurity firm FireEye after they detected anomalous activity related to their own internal systems.

Week-4: Cybersecurity Architecture

It was a catastrophic failure of supply chain security, build process integrity, access control, and threat detection.

Conceptual System Architecture of the Compromised Process

While the exact internal architecture isn't public, a conceptual representation of the relevant SolarWinds Orion build and update distribution process likely involved:


Conceptual SolarWinds Orion Build & Update Process
(Highlighting SUNBURST Compromise Point)
SolarWinds Internal Environment
Customer Environment
Build Servers / Orchestration (e.g., Jenkins, TeamCity) *** COMPROMISED ***
<< Malicious Code (SUNBURST) Injected Here (Step 2) >>
Artifact Repository
Update Distribution Servers
Customer Orion Instances
Code Signing Infrastructure (e.g., HSM)
Source Code Repository (e.g., Git)
Developers
1. Pull Source Code
3. Request Signing
4. Return Signed Binary Info
5. Push Signed Artifact (incl. SUNBURST)
6. Deploy Signed Update
7. Customer Downloads Update
Push/Pull Code

- **Source Code Repository:** Systems (like Git, SVN) holding the Orion platform's source code. Accessed by developers.
- **Build Servers/Orchestration:** Automated systems (e.g., Jenkins, TeamCity, Azure DevOps) responsible for retrieving source code, compiling it, running tests, and packaging the software. *This was the core compromised component.*
- **Code Signing Infrastructure:** Secure systems (potentially Hardware Security Modules - HSMs, or tightly controlled servers) holding the private keys used to digitally sign the compiled binaries. The compromised build process needed to interact with this system to get the malicious DLL signed.
- **Artifact Repository:** Storage for the compiled and signed software builds before distribution.
- **Update Distribution Servers:** Web servers and infrastructure accessible to customers to download Orion platform updates. These servers hosted the trojanized, signed updates.
- **Customer Orion Instances:** Customer-managed servers running the Orion platform, configured to check for and download updates from SolarWinds' distribution servers.

Week-4: Cybersecurity Architecture

The critical failure point was the **Build Server/Orchestration** stage, where attackers could inject code *before* the **Code Signing** stage, thus legitimizing the malware.

How Security Architectures Could Have Mitigated the Attack

No architecture guarantees prevention against such a sophisticated state-sponsored attack, but applying principles from various frameworks could have significantly increased the chances of prevention, detection, or containment.

a) Defense in Depth (DiD) Perspective

DiD could have provided multiple opportunities to disrupt the attack chain:

- **Network Layer:** Strict segmentation of the build environment from the general corporate network and the internet. Egress filtering from build servers to allow communication only with explicitly approved destinations (blocking C2 communication attempts by attackers *on* the build server). Micro-segmentation *within* the build pipeline (e.g., isolating build servers from code repositories except during checkout, isolating signing servers).
- **Endpoint Layer:** Hardening build servers (least privilege, minimal services, CIS benchmarks). Advanced EDR/XDR on build servers to detect anomalous processes, file modifications, or network connections associated with the build compromise or backdoor injection. Strict change control and file integrity monitoring (FIM) on build servers and source code.
- **Application Layer (Build Tools):** Security hardening of the build orchestration tools themselves (e.g., Jenkins). Input validation and sanitization if build scripts accept external parameters.
- **Data Layer:** Encryption and strict access controls on the source code repository and artifact repository. Protecting the code signing keys (e.g., using HSMs with strict access controls and auditing) is paramount.
- **Monitoring Layer:** Enhanced logging from build servers, source code repositories, signing infrastructure, and network devices, fed into a SIEM. Correlation rules are designed to detect anomalies in the build process (e.g.,

unexpected code checkouts, unusual build times, unexpected modules being compiled, unauthorized access attempts to signing keys). UEBA to detect anomalous developer or build service account behavior.

- **Recovery Layer:** While less about prevention, having secure, validated backups of the build environment and source code could aid investigation and recovery *after* detection.

DiD Failure Point Addressed: Lack of sufficient segmentation, monitoring, and hardening within the critical build environment.

b) Zero Trust Architecture (ZTA) Perspective

ZTA focuses on identity and explicit verification, which could have hindered the attackers at multiple points:

- **Identity Verification:** Strong MFA for all developers and administrators accessing the build environment. Use of machine identities (e.g., SPIFFE/SPIRE, managed identities) for build processes themselves, with tightly scoped permissions. Preventing the use of weak/shared credentials like `solarwinds123`.
- **Least Privilege Access:** Grant developers access only to the specific code repositories they need. Grant build service accounts only the minimum permissions required to check out code, compile, and request signing—*not* broad administrative access. Strict, audited, just-in-time access is available for any privileged operations on build or signing servers.
- **Device Trust:** Verifying the security posture of developer workstations and build servers before allowing access to code repositories or sensitive parts of the build pipeline. Compromised build servers might fail posture checks.
- **Micro-segmentation:** ZTA mandates granular segmentation. If applicable, network policies (or service mesh policies) would restrict build servers from communicating with anything other than necessary components (code repo, artifact repo, signing service via specific APIs). Prevent lateral movement from a compromised build server.

- **Assume Breach & Continuous Verification:** Every step in the build process requires authentication and authorization (e.g., the build service authenticates to the code repo, and the signing service authenticates to the signing service). Anomalous access patterns (e.g., a build server trying to access unexpected resources) would trigger alerts. Continuous monitoring of identity logs and access patterns is key.
- **API Security:** Treating interactions between build components (e.g., build orchestrator to signing service) as API calls requiring strong authentication and authorization.

ZTA Failure Point Addressed: Implicit trust within the build environment, weak authentication, excessive privileges for build processes/accounts.

c) Secure Software Development Lifecycle (SSDLC) Perspective

SSDLC is perhaps the *most directly relevant* architecture for preventing the core code injection:

- **Threat Modeling:** Explicitly threat modeling the build pipeline itself to identify risks like code injection, signing key compromise, dependency manipulation, etc.
- **Secure Build Environment:** Hardening build servers, securing the CI/CD platform, and minimizing tools/dependencies on build servers.
- **Source Code & Dependency Security:** SAST scanning of source code (might not find sophisticated backdoors easily, but could find precursor flaws). Rigorous Software Composition Analysis (SCA) to check third-party dependencies used *in the build process itself*.
- **Build Integrity & Reproducibility:** Implementing checks to ensure the code being compiled matches the code in the repository (e.g., hashing source files before build). Striving for reproducible builds where the same source code always produces bit-for-bit identical binaries, making unauthorized changes easier to detect.
- **Artifact Signing & Verification:** While the malware *was* signed, a mature SSDLC includes *validating* the signing process. This could involve checks like:

ensuring only expected artifacts are presented for signing, multi-party approval for signing critical components, auditing signing logs for anomalies, potentially performing post-build/pre-distribution binary analysis, or diffing against known-good builds (very hard for complex software).

- **Secure Supply Chain Practices:** Generating and verifying Software Bills of Materials (SBOMs). Implementing frameworks like SLSA (Supply-chain Levels for Software Artifacts) to provide verifiable evidence about build provenance and integrity.
- **Secrets Management:** Securely managing credentials used by the build pipeline (e.g., access tokens for code repos, credentials for artifact repos) using vaults, not hardcoded or easily accessible.

SSDLC Failure Point Addressed: Lack of sufficient security controls, integrity checks, and validation *within the build and release process itself*.

d) Zero Knowledge Architecture (ZKA) Perspective

ZKA's primary goal is to protect user data confidentiality from service providers. Its direct applicability to preventing the SolarWinds *supply chain compromise* is limited. However, ZKA *principles* related to cryptographic key protection could be relevant:

- **Code Signing Key Protection:** While not strictly ZKA, applying principles of minimizing trust and knowledge could involve using HSMs or confidential computing environments for code signing where the raw private key material is never directly accessible even to privileged administrators or the build system itself. The build system would submit the hash of the binary to the secure environment, which performs the signing operation internally without exposing the key. This makes direct key theft harder and forces attackers to compromise the *process* of submitting hashes, which might be easier to detect via ZTA/ASA.
- **Secrets Management in Build:** Applying zero-knowledge principles (or more practically, robust secrets management with least privilege) to protect sensitive credentials or configuration data used *within* the build pipeline, making it harder

for attackers on a compromised build server to steal credentials needed for lateral movement or persistence.

ZKA Failure Point Addressed: Indirectly, by potentially enhancing the protection of the critical code signing keys, making the signing of malicious code harder without compromising the signing infrastructure itself.

e) Adaptive Security Architecture (ASA) Perspective

ASA focuses on continuous monitoring, behavioral analysis, and automated response, which is crucial for *detecting* sophisticated attacks like SUNBURST:

- **Continuous Monitoring & Telemetry:** Collecting detailed logs and telemetry from *all* relevant sources: build servers (process execution, file changes, network connections), source code repositories (checkout/commit activity), identity systems (logins, privilege changes), network sensors (especially egress traffic), and endpoint agents (EDR/XDR).
- **Behavioral Analytics (UEBA/NBA):** Establishing baselines for normal build server behavior, developer activity, and network traffic patterns. Detecting deviations such as:
    - Anomalous processes running on build servers.
    - Unexpected modifications to source code files or build scripts *during* the build.
    - Build servers making unusual outbound network connections (potential C2 or data exfiltration by attackers *on* the build server).
    - Developer accounts are performing unusual actions or accessing systems outside their norm.
- **Threat Intelligence Integration:** Ingesting threat intelligence feeds about attacker TTPs, known malicious IPs/domains, and indicators of compromise related to supply chain attacks or specific threat actors (like Cozy Bear/APT29).
- **Context-Aware Policies & Risk Scoring:** Dynamically adjusting security postures based on detected anomalies. For example, increasing scrutiny or

requiring additional verification for builds exhibiting unusual behavior, or temporarily isolating suspicious build servers.

- **Automated Response (SOAR):** Pre-defined playbooks to automatically respond to high-confidence alerts, such as:
    - Isolating a build server exhibiting malicious activity from the network.
    - Revoking credentials associated with suspicious activity.
    - Blocking known malicious C2 domains detected in egress traffic.
    - Triggering enhanced logging or forensic capture on suspect systems.

ASA Failure Point Addressed: Lack of sufficient monitoring, behavioral detection, and rapid response capabilities to identify the subtle compromise and malicious activities within the SolarWinds environment *before* the trojanized update was widely distributed.

Conclusion: The SolarWinds attack highlights the need for a multi-faceted security strategy. While SSDLC is paramount for preventing the initial supply chain compromise, DiD, ZTA, and ASA are crucial for protecting the environment where development occurs and for detecting and responding to compromises when prevention fails.