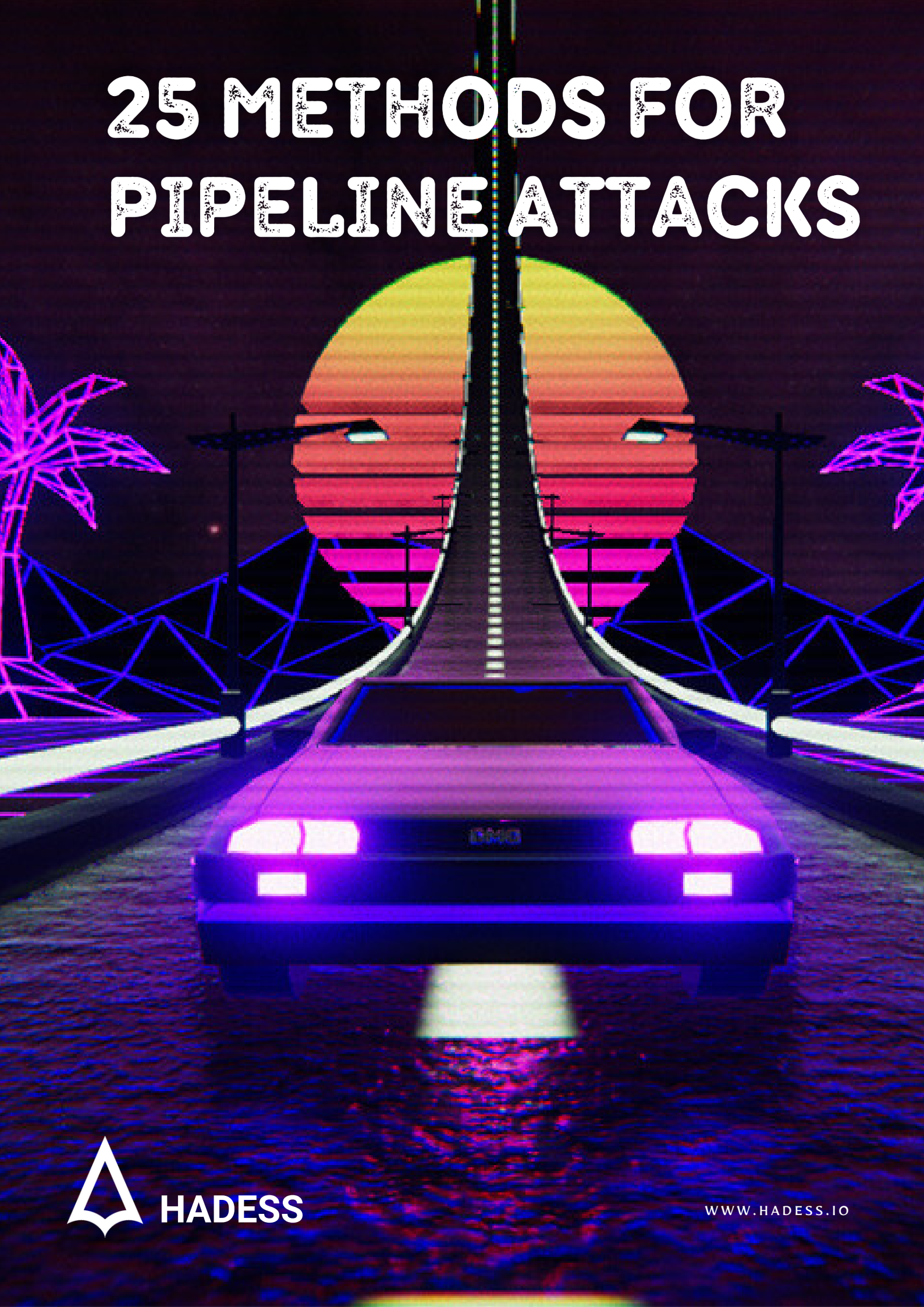
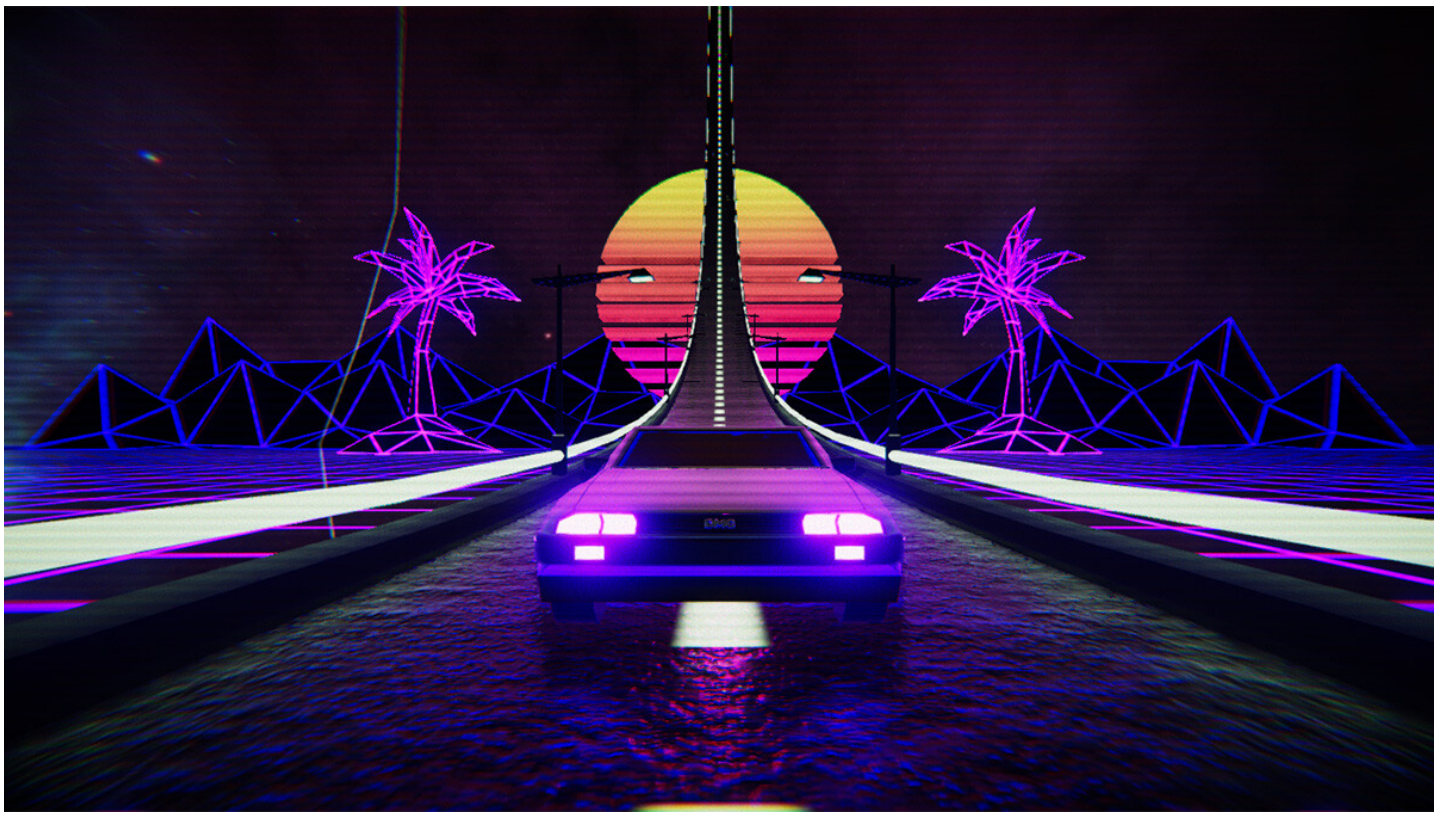
25 METHODS FOR PIPELINE ATTACKS



HADESS

WWW.HADESS.IO

25 Methods for Pipeline Attacks(RTC0011)



Add Approver using Admin permission on CI

1. Unauthorized Approver Addition

In this scenario, an attacker with admin permissions on the CI pipeline adds an unauthorized user as an approver, potentially bypassing necessary security checks.

```
# Add unauthorized user as an approver
ci-tool add-approver --pipeline pipeline-name --user unauthorized-user
```

1. Exploiting Weak Authentication

In this scenario, an attacker leverages weak authentication mechanisms to gain admin access to the CI tool and manipulate approver settings.

```
# Brute-force attack to guess admin credentials
ci-tool login --username admin --password weakpassword

# Add unauthorized user as an approver
ci-tool add-approver --pipeline pipeline-name --user unauthorized-user
```

1. API Token Misuse

```
# Use stolen API token to add unauthorized user as an approver
ci-tool --api-token stolen-token add-approver --pipeline pipeline-name --user
unauthorized-user
```

Dumping Env Variables in CI/CD

1. Code Review with Malicious Payload

```
# Dump environment variables during the CI/CD pipeline execution
echo "printenv" >> main.py
```

In this scenario, an attacker submits a code change to a legitimate repository that, when executed in the CI/CD pipeline, dumps environment variables containing sensitive information.

1. Exploiting Debug Mode

```
# Enable debug mode to print environment variables
ci-tool --debug execute-pipeline
```

In this scenario, an attacker leverages the debug mode of the CI/CD pipeline to expose environment variables containing sensitive information.

1. Code Injection via Build Steps

```
# Inject malicious command to dump environment variables
ci-tool run-step --command "echo $ENV_VARIABLES"
```

In this scenario, an attacker injects malicious commands as build steps in the CI/CD pipeline, which results in the dumping of environment variables.

Access to Secret Manager from CI/CD kicked by different repository

1. Exploiting Weak Permissions

```
# Retrieve secrets from the secret manager
ci-tool get-secrets --repository malicious-repo --secret-manager secret-
manager-name
```

In this scenario, an attacker gains access to a CI/CD pipeline job that interacts with a secret manager and exploits weak permissions to retrieve sensitive credentials.

1. Code Injection in CI Pipeline

```
# Inject code to capture secrets during the CI pipeline execution
echo "echo $SECRET_VARIABLE" >> main.py
```

In this scenario, an attacker submits a malicious code change to a legitimate repository, which, when executed in the CI pipeline, captures secrets or leverages them to perform unauthorized actions.

1. Exploiting Misconfigured Environment Variables

```
# Retrieve secrets using misconfigured environment variables
ci-tool get-secrets --repository legitimate-repo --secret-manager
$MISCONFIGURED_VARIABLE
```

In this scenario, an attacker identifies misconfigured environment variables within the CI/CD pipeline and leverages them to gain unauthorized access to the secret manager.

please make three real example scenario for Get credential from CI/CD Admin Console in pipeline attacks in devsecops with tools commands and codes in one liner command pattern stage

getting credentials from a CI/CD admin console

1. AWS CLI

```
export AWS_ACCESS_KEY_ID=$(aws ssm get-parameter --name /path/to/access_key --with-decryption --query Parameter.Value --output text)
export AWS_SECRET_ACCESS_KEY=$(aws ssm get-parameter --name /path/to/secret_key --with-decryption --query Parameter.Value --output text)
pipeline-stage-command
```

Retrieve AWS access and secret keys from the CI/CD admin console and pass them as environment variables to the pipeline stage

1. Vault CLI

```
export SECRET=$(vault kv get -format=json secret/path/to/secret | jq -r '.data.key')
pipeline-stage-command
```

Retrieve a secret from HashiCorp Vault and pass it as an environment variable to the pipeline stage

1. Kubernetes CLI

```
kubectl create secret generic my-secret --from-literal=username=<username> --
from-literal=password=<password>
kubectl create volume secret my-secret-volume --secret=my-secret
pipeline-stage-command --volume=my-secret-volume:/secrets
```

Retrieve a Kubernetes secret from the admin console and pass it as a mounted volume to the pipeline stage

Modify the configuration of Production environment

1. Unauthorized Configuration Modification

```
# Modify the production environment configuration file
echo "MALICIOUS_SETTING = True" >> production.config
```

In this scenario, an attacker gains access to the CI/CD pipeline and modifies the configuration of the production environment, potentially introducing security vulnerabilities or disrupting the system.

1. Exploiting Misconfigured Deployment Scripts

```
# Exploit misconfigured deployment script to modify production configuration
ci-tool run-deployment-script --script "sed -i 's/ALLOW_PUBLIC_ACCESS =
False/ALLOW_PUBLIC_ACCESS = True/' production.config"
```

In this scenario, an attacker identifies a misconfigured deployment script in the CI/CD pipeline that allows unauthorized modification of the production environment configuration.

1. Social Engineering and Unauthorized Access

```
# Social engineering attack to gain access and modify the production
configuration
ci-tool login --username admin --password attacker-password
ci-tool modify-config --environment production --setting MALICIOUS_SETTING --
value True
```

In this scenario, an attacker employs social engineering techniques to gain unauthorized access to the CI/CD pipeline and modifies the production environment configuration.

Insecure Configuration Management

1. Using Trivy:

```
curl -X POST -d "repo_url=malicious_repo.git; rm -rf /"
https://example.com/deploy
```

Code:

```
import subprocess

def deploy(repo_url):
    # Insecure configuration, command injection vulnerability
    command = f"git clone {repo_url}"
    subprocess.call(command, shell=True)
```

In this example, the `deploy` function is part of a deployment script. The `repo_url` parameter is expected to be a valid Git repository URL. However, due to improper input validation, an attacker can inject arbitrary commands by appending them to the `repo_url`. In this case, the attacker adds a command (`rm -rf /`) to delete the entire filesystem.

1. Environment Variable Disclosure

```
curl https://example.com/api/config
```

Code:

```
import os

def get_api_key():
    # Insecure configuration, sensitive information exposure
    return os.environ.get('API_KEY')
```

In this example, the `get_api_key` function retrieves an API key from an environment variable (`API_KEY`). However, the pipeline configuration has a vulnerability that allows unauthorized access to environment variables. An attacker can make a request to the `/api/config` endpoint to retrieve sensitive information, such as the API key.

1. Insecure Credential Storage

```
cat ~/.aws/credentials
```

Code:

```
import boto3

def list_buckets():
    # Insecure configuration, sensitive information exposure
    session = boto3.Session()
    s3 = session.client('s3')
    return s3.list_buckets()
```

In this example, the `list_buckets` function uses the AWS SDK to interact with Amazon S3. The code relies on the default credential lookup behavior of the SDK, which checks various configuration files for AWS credentials. However, the pipeline has a security flaw where the AWS credentials are stored in an insecure manner. An attacker can execute a command (`cat ~/.aws/credentials`) to retrieve the AWS credentials and potentially gain unauthorized access to the AWS resources.

Weak Authentication and Authorization

1. Using Trivy:

```
trivy -q -f json <container_name>:<tag> | jq '.[] | select(.Vulnerabilities != null)'
```

This command uses Trivy, a vulnerability scanner for containers, to scan a specific container image (`<container_name>:<tag>`) for vulnerabilities. The `-q` flag suppresses the output, and the `-f json` flag formats the output as JSON. The command then uses `jq` to filter the results and display only the vulnerabilities found.

1. Using Clair-scanner:

```
clair-scanner --ip <container_ip> --report <report_output_file>
```

This command utilizes Clair-scanner, a tool that integrates with the Clair vulnerability database, to scan a running container (`<container_ip>`) and generate a report (`<report_output_file>`) containing the vulnerabilities found.

1. Using kube-hunter:

```
kube-hunter --remote <cluster_address> | grep -B 5 "Critical:"
```

This command employs kube-hunter, a Kubernetes penetration testing tool, to scan a remote Kubernetes cluster (`<cluster_address>`) for security vulnerabilities. The output is then piped to `grep` to filter and display only the critical vulnerabilities found.

Insecure CI/CD Tools

1. Using trufflehog:

```
trufflehog --regex --entropy=True https://example.com/git-repo.git
```

This command utilizes TruffleHog, a sensitive data scanner for Git repositories. It scans the specified Git repository URL (`https://example.com/git-repo.git`) using regex pattern matching and entropy analysis to identify potentially sensitive

information, such as API keys, passwords, and other secrets stored in the repository.

1. Using circleci:

```
docker run -v $(pwd):/src -w /src -t circleci/circleci-cli:latest circleci config pack .circleci > config.yml && circleci local execute -c config.yml
```

This command uses the CircleCI CLI tool to simulate the execution of a CircleCI configuration file (`config.yml`) locally. It packs the configuration file using `circleci config pack`, and then executes the packed configuration file using `circleci local execute`. This can help identify misconfigurations, vulnerabilities, or insecure practices within the CI/CD pipeline defined in the CircleCI configuration file.

1. Using git:

```
git clone https://github.com/username/repo.git && cd repo && git log --pretty=format:%H -n 1 | xargs -I {} git checkout {} && git grep -iE "(password|secret|token)" .
```

This command clones a Git repository (`https://github.com/username/repo.git`) and navigates to the repository directory. It retrieves the latest commit hash (`git log --pretty=format:%H -n 1`) and uses it to check out that specific commit (`git checkout {}`). Finally, it performs a case-insensitive search (`git grep -iE`) for common sensitive keywords like “password,” “secret,” or “token” within the repository code. This helps identify potential security issues related to sensitive information being exposed in the CI/CD pipeline.

Lack of Secure Coding Practices

1. Using snyk:

```
snyk test --all-projects --all-sub-projects
```

This command uses Snyk, a popular security scanning tool, to test all projects and sub-projects in the current directory. Snyk scans the codebase and its dependencies, checking for known vulnerabilities, insecure coding practices, and other security issues. This helps identify and address security weaknesses introduced by insecure coding practices.

1. Using bandit:

```
bandit -r /path/to/source-code
```

This command utilizes Bandit, a security tool specifically designed for Python, to perform static code analysis on the source code located at `/path/to/source-code`. Bandit checks for common security issues in Python code, such as the use of insecure cryptographic algorithms, improper handling of user input, and potential code injection vulnerabilities. It helps identify insecure coding practices that could lead to security vulnerabilities.

1. Using npm:

```
npm audit --registry=https://registry.npmjs.org/
```

This command uses the `npm audit` command, which is built into the Node Package Manager (npm), to perform a security audit of the dependencies in a Node.js project. It checks the project's dependencies against the National Vulnerability Database (NVD) and provides a report highlighting any known vulnerabilities and insecure coding practices. This helps identify and address security issues introduced by insecurely implemented or outdated dependencies.

Insecure Third-Party Dependencies

1. Using OWASP-Dependency-Check:

```
OWASP-Dependency-Check --scan <path-to-project> --format HTML
```

This command uses OWASP Dependency-Check, a widely-used tool for identifying known vulnerabilities in project dependencies. It scans the project located at `<path-to-project>`, analyzes the dependencies, and generates an HTML report (`--format HTML`) that highlights any insecure or vulnerable third-party components used in the project. This helps identify and address security risks arising from insecure third-party dependencies.

1. Using Retire:

```
Retire.js -c -s <path-to-project>
```

This command utilizes Retire.js, a tool for detecting vulnerable JavaScript libraries and outdated dependencies. It scans the project located at `<path-to-project>` and checks the JavaScript files for known vulnerabilities in commonly used libraries. The `-c` option shows the full file path, and the `-s` option suppresses non-vulnerable lines. This helps identify and address security issues arising from insecure or outdated third-party JavaScript dependencies.

1. Using bundle-audit:

```
bundle-audit check --update
```

This command uses bundle-audit, a Ruby-based tool, to check for known vulnerabilities in Ruby gems used in a project. The `check` command scans the project's Gemfile.lock and checks for any insecure or vulnerable gems. The `--update` option updates the vulnerability database before performing the check. This helps identify and address security risks arising from insecure or outdated third-party Ruby dependencies.

Insufficient Testing

1. Using OWASP ZAP:

```
owasp-zap-cli -s -t http://example.com
```

This command uses OWASP ZAP (Zed Attack Proxy) Command Line Interface (CLI) to perform automated security testing on a target website (`http://example.com`). The `-s` option starts the scan, and the `-t` option specifies the target URL. OWASP ZAP scans the target for common security vulnerabilities, such as cross-site scripting (XSS) and SQL injection. This helps identify potential security weaknesses resulting from insufficient testing.

1. Using goss:

```
goss validate -f /path/to/goss.yaml
```

This command utilizes Goss, a tool for validating system configurations and testing infrastructure. It runs tests defined in the `goss.yaml` file (`-f /path/to/goss.yaml`) to verify the expected state of the system. Goss validates various aspects, including file contents, process status, and network connectivity. This helps ensure that the system is properly configured and tested, exposing any insufficiencies in the testing process.

1. Using junit-viewer:

```
junit-viewer --results /path/to/test-results.xml --output-dir /path/to/output
```

This command uses JUnit-Viewer, a tool for visualizing JUnit test results. It takes the JUnit XML test results file (`--results /path/to/test-results.xml`) and generates an HTML report in the specified output directory (`--output-dir /path/to/output`). This allows for easy visualization and analysis of test results, helping to identify areas where testing might be insufficient or inadequate.

Insecure Build and Deployment Processes

1. Using Trivy:

```
trivy image <image-name>:<tag>
```

This command uses Trivy, a vulnerability scanner for container images, to scan a specific Docker image (`<image-name>:<tag>`) for known vulnerabilities. Trivy analyzes the image's layers and dependencies, checking for security issues in packages and libraries. This helps identify vulnerabilities in the image that could be introduced during the build and deployment processes.

1. Using kube-hunter:

```
kube-hunter --remote
```

This command uses kube-hunter, a Kubernetes penetration testing tool, to assess the security of a Kubernetes cluster. The `--remote` option allows kube-hunter to scan a remote cluster for security vulnerabilities and misconfigurations related to the build and deployment processes. It tests various attack vectors and highlights potential weaknesses in the cluster's configuration.

1. Using owtf:

```
owtf -s "target_host"
```

This command utilizes OWTF (Open Web Application Security Project Testing Framework), a penetration testing tool, to perform security assessments on a target host. OWTF combines various tools and techniques to evaluate the security of web applications and underlying systems. By targeting the host involved in the build and deployment processes, OWTF can identify vulnerabilities and weaknesses that might exist within the build and deployment pipeline.

Exposed Credentials

1. Using gitleaks:

```
gitleaks --repo=https://github.com/username/repo.git
```

This command uses Gitleaks, a tool for detecting secrets and credentials in Git repositories, to scan a specific repository (`https://github.com/username/repo.git`). Gitleaks scans the repository's commit history and identifies any exposed credentials, such as API keys, passwords, or sensitive information. This helps identify potential security risks resulting from exposed credentials within the codebase.

1. Using trufflehog:

```
trufflehog --regex --entropy=True https://example.com/git-repo.git
```

This command utilizes TruffleHog, a sensitive data scanner for Git repositories. It scans the specified Git repository URL (`https://example.com/git-repo.git`) using regex pattern matching and entropy analysis to identify potentially exposed credentials and other sensitive information. TruffleHog helps detect security risks arising from exposed credentials within the repository.

1. Using nuclei:

```
nuclei -l /path/to/targets.txt -t /path/to/templates-dir -t secrets-disclosure -o output.txt
```

This command uses Nuclei, a powerful vulnerability scanner, to perform targeted scanning for secrets disclosure vulnerabilities. The `-l` option specifies the file containing a list of target URLs in `/path/to/targets.txt`. The `-t` option points to the directory path containing the Nuclei templates, including the `secrets-disclosure` template that checks for exposed credentials. The `-o` option specifies the output file where the results will be saved.

Insufficient Monitoring and Logging

1. Using lynis:

```
lynis audit system
```

This command uses Lynis, an open-source security auditing tool, to perform an audit of the system. Lynis checks various aspects of the system's security, including monitoring and logging configurations. It identifies potential weaknesses in log management, monitoring solutions, and auditing mechanisms. This helps assess the system's monitoring and logging practices for potential security gaps.

1. Using auditd:

```
auditd -f -l
```

This command utilizes Auditd, the Linux audit framework, to view the current audit rules and configuration. The `-f` option specifies that the output should be formatted for easy readability. Auditd provides monitoring and logging capabilities by generating logs of system events. By examining the audit rules and configuration, one can determine if they are set up to monitor and log the necessary security-relevant events.

1. Using splunk:

```
splunk search 'sourcetype=access_*' earliest=-1d latest=now
```

This command utilizes Splunk, a popular log management and analysis platform, to search for access logs (`sourcetype=access_*`) from the past 24 hours (`earliest=-1d latest=now`). By querying the access logs, it helps assess if sufficient monitoring and logging are in place. This can identify any gaps in log collection, retention, or analysis, highlighting potential weaknesses in monitoring and logging practices.

Misconfigured Access Controls

1. Misconfigured Access Control in CI/CD Pipeline Secrets

```
gitleaks --repo=https://github.com/your-repo.git --report=git-secrets-report.txt
```

This command utilizes GitLeaks to scan a Git repository (`your-repo.git`) for potential secrets that might have been mistakenly committed. Misconfigured access controls in the CI/CD pipeline could allow unauthorized access to sensitive information. The `--report` parameter specifies the file where the findings will be saved.

1. Misconfigured IAM Policies in Cloud Deployment

```
cloudsploit scan --scan-id=1234 --scan-type=iam --output-file=iam-policy-report.json
```

Here, CloudSploit is used to perform an IAM policy scan in a cloud deployment. Misconfigured access controls in IAM policies can lead to unauthorized access to cloud resources. The `--scan-id` parameter assigns a unique identifier to the scan, while `--scan-type` specifies the IAM policy scan. The `--output-file` parameter defines the file where the scan results will be stored.

1. Misconfigured Role-Based Access Control (RBAC) in Kubernetes Cluster

```
kube-hunter --remote --report kube-hunter-report.txt
```

This command runs kube-hunter in remote mode to assess the security of a Kubernetes cluster. It scans for potential misconfigurations in role-based access controls (RBAC), which can result in unauthorized access to critical resources. The `--remote` flag instructs kube-hunter to scan a remote cluster, and the `--report` parameter specifies the file where the findings will be saved.

Insecure Configurations

1. Insecure Secrets Management in CI/CD Pipeline

```
trufflehog --regex --entropy=True https://github.com/your-repo.git > insecure-secrets.txt
```

1. This command uses Trufflehog to scan a Git repository (`your-repo.git`) for insecurely stored secrets, such as API keys and passwords. Insecure configurations in secrets management can lead to unauthorized access to sensitive information. The `--regex` flag enables regular expression matching, and the `--entropy` flag enables entropy checks. The output is saved to the `insecure-secrets.txt` file.

2. Insecure Docker Image Configurations

```
anchore-cli analyze docker://your-image:tag --image-id=your-image-id --all-tags --report insecure-docker-image-report.json
```

1. Here, Anchore Engine is used to analyze a Docker image (`your-image:tag`) for insecure configurations. Insecure settings in Docker images can introduce vulnerabilities into the pipeline. The `--image-id` parameter specifies the image ID, and `--all-tags` instructs Anchore to analyze all available tags. The analysis results are saved in the `insecure-docker-image-report.json` file.

2. Insecure Infrastructure-as-Code (IaC) Configurations

```
tfsec your-iac-directory --format=json --output=tfsec-results.json
```

1. This command utilizes tfsec to scan an Infrastructure-as-Code (IaC) directory for insecure configurations. Insecure IaC settings can lead to misconfigured infrastructure and potential vulnerabilities. The `--format` parameter specifies the output format as JSON, and the `--output` parameter defines the file where the scan results will be saved (`tfsec-results.json`).

Vulnerability Management

1. Vulnerability Scanning of Docker Images in CI/CD Pipeline

```
clair-scanner --ip <IP_ADDRESS> --report=clair-report.json <DOCKER_IMAGE>
```

1. This command uses Clair to scan a Docker image (`<DOCKER_IMAGE>`) for vulnerabilities in the CI/CD pipeline. Clair is an open-source vulnerability scanner designed for containerized environments. The `--ip` parameter specifies the IP address of the Clair server, and the `--report` parameter defines the file where the scan results will be saved (`clair-report.json`).

2. Static Application Security Testing (SAST) in Pipeline

```
bandit -r <SOURCE_DIRECTORY> -f json -o bandit-report.json
```

1. Here, Bandit is used for static application security testing (SAST) in the pipeline. It scans the source code directory (`<SOURCE_DIRECTORY>`) for potential security issues. The `-f json` parameter specifies the output format as JSON, and the `-o` parameter defines the file where the scan results will be saved (`bandit-report.json`).

2. Dependency Scanning in CI/CD Pipeline

```
dependency-check.sh --scan <PROJECT_DIRECTORY> --format JSON --out
dependency-check-report.json
```

1. This command runs OWASP Dependency-Check to scan a project directory (`<PROJECT_DIRECTORY>`) for known vulnerabilities in dependencies. Dependency-Check identifies and reports any outdated or vulnerable libraries. The `--format` parameter specifies the output format as JSON, and the `--out` parameter defines the file where the scan results will be saved (`dependency-check-report.json`).

Inadequate Secrets Management

1. Insecure Storage of Secrets in Source Code Repositories

```
gitleaks --repo=https://github.com/your-repo.git --report=secrets-leak-
report.txt
```

1. This command uses GitLeaks to scan a Git repository (`your-repo.git`) for potential secrets that might have been mistakenly committed. Inadequate secrets management can result in sensitive information being exposed in source code repositories. The `--report` parameter specifies the file where the findings will be saved.

2. Insecure Storage of Secrets in Infrastructure-as-Code (IaC) Templates

```
tfsec your-iac-directory --no-color --output=iac-secrets-report.txt
```

1. Here, tfsec is used to scan an Infrastructure-as-Code (IaC) directory for insecure storage of secrets. Inadequate secrets management in IaC templates can lead to sensitive information being stored in plain text or insecurely. The `--no-color` flag disables colored output, and the `--output` parameter defines the file where the scan results will be saved.

2. Insecure Secrets Configuration in CI/CD Pipeline

```
snyk test --all-projects --json > pipeline-secrets-report.json
```

This command runs Snyk to test all projects in a CI/CD pipeline for insecure secrets configurations. Inadequate secrets management in the pipeline can expose sensitive information to unauthorized access. The `--all-projects` parameter instructs Snyk to test all projects, and the `--json` flag specifies the output format as JSON. The scan results are saved in the `pipeline-secrets-report.json` file.

Insecure Third-Party Integrations

1. Insecure API Integration in CI/CD Pipeline

```
zap-cli --zap-url http://localhost -p 8080 -c 'quick-scan -s -t
https://api.example.com' -r insecure-api-integration-report.html
```

This command utilizes OWASP ZAP's command-line interface (zap-cli) to perform a quick scan on the API endpoint (`https://api.example.com`) integrated into the CI/CD pipeline. Insecure third-party API integrations can introduce vulnerabilities. The `-s` flag enables spidering of the target API, and the `-r` parameter defines the file where the scan results will be saved in HTML format.

1. Insecure Library or Component in CI/CD Pipeline

```
snyk test --all-projects --json > insecure-library-report.json
```

1. Here, Snyk is used to test all projects in a CI/CD pipeline for insecure libraries or components. Insecure third-party dependencies can introduce vulnerabilities into the pipeline. The `--all-projects` parameter instructs Snyk to test all projects, and the `--json` flag specifies the output format as JSON. The scan results are saved in the `insecure-library-report.json` file.

2. Insecure Cloud Provider Integration in Pipeline

```
cloudgoat create --template pipeline --duration 2h
```

This command uses CloudGoat to create a simulated insecure cloud provider integration in the pipeline. CloudGoat is a vulnerable-by-design AWS deployment tool. Insecure third-party integrations with cloud providers can result in unauthorized access or data breaches. The `--template` parameter specifies the pipeline template, and the `--duration` parameter defines the duration of the simulation (2 hours in this example).

Weak Access Controls

1. Weak Access Controls on CI/CD Pipeline Configuration

```
arachni http://your-ci-cd-pipeline.com --plugin=login_script --report-save-
path=weak-access-controls-report.afr
```

This command uses Arachni to scan the CI/CD pipeline configuration website (`http://your-ci-cd-pipeline.com`) for weak access controls. Weak access controls on the CI/CD pipeline configuration can lead to unauthorized access and potential attacks. The `--plugin=login_script` parameter enables the login script plugin, and the `--report-save-path` parameter defines the file where the scan results will be saved in Arachni format.

1. Weak Access Controls on Cloud Infrastructure

```
prowler -c all -r us-west-2 > weak-access-controls-report.txt
```

Here, Prowler is used to perform a security assessment on the cloud infrastructure in the US West 2 region. Weak access controls in cloud infrastructure can expose sensitive resources to unauthorized access. The `-c all` parameter scans all available checks, and the `-r` parameter specifies the region. The scan results are saved in the `weak-access-controls-report.txt` file.

1. Weak Access Controls in Kubernetes RBAC

```
kube-hunter --remote --report weak-access-controls-report.txt
```

This command runs kube-hunter in remote mode to identify weak access controls in a Kubernetes cluster. Weak RBAC configurations in Kubernetes can result in unauthorized access to critical resources. The `--remote` flag instructs kube-hunter to scan a remote cluster, and the `--report` parameter defines the file where the findings will be saved (`weak-access-controls-report.txt`).

Insider Threats

1. Monitoring Code Changes for Suspicious Activity

```
git log --author="John Doe" --since="1 month ago" --oneline
```

This command lists the Git commit history for a specific author, such as “John Doe,” within the last month. By monitoring code changes associated with individuals who may pose an insider threat, you can identify suspicious activity that goes beyond regular development activities. However, it’s essential to consider privacy and legal implications when monitoring employee activities.

1. Auditing System and Application Logs

```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        {"match": { "user": "John Doe" }},
        {"range": { "@timestamp": { "gte": "now-1d", "lte": "now" }}}
      ]
    }
  }
}
```

This Elasticsearch query retrieves logs from the last 24 hours for a specific user, such as “John Doe.” By implementing a centralized logging system like ELK Stack and querying relevant logs, you can search for suspicious activities or unusual patterns that may indicate insider threats.

1. Implementing User Behavior Analytics (UBA)

```
metron_monitor.sh -r insider_threat_rules.json
```

This command runs Apache Metron with a set of insider threat detection rules specified in the `insider_threat_rules.json` file. User Behavior Analytics (UBA) tools like Apache Metron can analyze user activities, system logs, and network traffic to detect anomalous behavior that may indicate insider threats. The specific rules and configurations within Metron can be tailored to your organization’s needs.

Lack of Build Integrity

1. Checking Code Integrity with Hash Verification

```
md5sum your-file.zip
```

This command calculates the MD5 hash of a file (`your-file.zip`). By comparing the calculated hash with the expected hash, you can verify the integrity of the file. This technique can be used to detect unauthorized modifications or tampering of code or build artifacts during the pipeline.

1. Verifying Digital Signatures of Builds

```
gpg --verify your-build.tar.gz.sig your-build.tar.gz
```

This command uses GnuPG (GPG) to verify the digital signature of a build (`your-build.tar.gz`) by comparing it to the corresponding signature file (`your-build.tar.gz.sig`). Digital signatures provide assurance of build integrity and authenticity. If the signature verification fails, it indicates a potential lack of build integrity.

1. Checking Docker Image Integrity with Image Digest

```
docker inspect --format='' your-image:tag
```

This command inspects a Docker image (`your-image:tag`) and retrieves the image digest. The image digest is a hash that uniquely identifies the image content. By comparing the digest with the expected value, you can ensure the integrity of the Docker image. A mismatch indicates a lack of build integrity or unauthorized modifications.

Inadequate Testing

1. Inadequate Unit Testing in Pipeline

```
pytest --cov=your_module tests/
```

This command runs pytest to execute unit tests for a specific module (`your_module`) in the pipeline. Inadequate unit testing can result in undetected code vulnerabilities and weaknesses. The `--cov` parameter enables code coverage analysis, providing insights into the adequacy of test coverage.

1. Insufficient Security Testing in Pipeline

```
zap-cli --zap-url http://localhost -p 8080 -c 'spider -r http://your-web-app.com && active-scan -r http://your-web-app.com' -x scan-results.html
```

This command uses OWASP ZAP's command-line interface (zap-cli) to perform automated security testing on a web application (`http://your-web-app.com`) in the pipeline. Insufficient security testing can lead to vulnerabilities being overlooked. The `spider` command crawls the application, and the `active-scan` command performs an active scan. The `-x` parameter specifies the file to export the scan results in HTML format.

1. Limited Penetration Testing in Pipeline

```
nmap -p 1-65535 -sV -sS -T4 your-ip-address
```

This command uses Nmap to perform a comprehensive port scan with service version detection on a specified IP address (`your-ip-address`). Limited penetration testing can miss critical vulnerabilities and weaknesses. The `-p` parameter specifies the port range, `-sV` enables service version detection, and `-sS` performs a TCP SYN scan. The `-T4` parameter sets the scan speed to a moderate level.

Insufficient Monitoring and Logging

1. Insufficient Monitoring of Web Application Logs

```
GET /your-web-app-logs*/_search?q=response_code:500
```

This Elasticsearch query retrieves logs related to HTTP 500 errors in a specific index (`your-web-app-logs*`) for a web application. Insufficient monitoring of web application logs can result in missed indications of attacks or vulnerabilities. By customizing the query, you can search for specific log events or patterns that may indicate security issues.

1. Inadequate Monitoring of Infrastructure Events

```
aws cloudtrail lookup-events --lookup-attributes
AttributeKey=EventName,AttributeValue=StopInstances
```

This AWS CLI command queries the CloudTrail service to lookup events related to stopping EC2 instances. Insufficient monitoring of infrastructure events can lead to missed indicators of unauthorized access or malicious activities. By specifying different `AttributeKey` and `AttributeValue` pairs, you can search for specific events relevant to your infrastructure.

1. Insufficient Real-time Monitoring of Network Traffic

```
suricata -c /etc/suricata/suricata.yaml -r your-pcap-file.pcap -l
/var/log/suricata/
```

This command runs Suricata with a specified configuration file (`/etc/suricata/suricata.yaml`) to analyze network traffic captured in a PCAP file (`your-pcap-file.pcap`). Insufficient real-time monitoring of network traffic can result in missed indicators of network-based attacks. By examining the logs generated in the specified directory (`/var/log/suricata/`), you can identify potential security events or anomalies.

Lack of Compliance and Governance

1. Lack of Compliance in CI/CD Pipeline Configuration

```
bandit -r your-source-code-directory -ll -f json -o compliance-report.json
```

This command runs Bandit, a SAST tool, to scan the source code directory (`your-source-code-directory`) for compliance-related issues. Lack of compliance in the pipeline configuration can lead to security vulnerabilities. The `-ll` flag enables long description output, the `-f json` parameter specifies the output format as JSON, and the `-o` parameter defines the file where the compliance scan results will be saved.

1. Insufficient Governance of Cloud Resources

```
aws configservice describe-compliance-by-resource --resource-type
AWS::EC2::Instance --compliance-types NON_COMPLIANT --output json >
governance-report.json
```

This AWS CLI command queries AWS Config to retrieve compliance information for AWS EC2 instances. Insufficient governance of cloud resources can result in non-compliant configurations and security risks. The `--resource-type` parameter specifies the resource type, the `--compliance-types` parameter filters for non-compliant resources, and the `--output json` flag formats the output as JSON. The results are saved in the `governance-report.json` file.

1. Inadequate Logging and Monitoring for Compliance

```
GET /your-logs-index/_search?q=compliance_status:failed
```

This Elasticsearch query retrieves logs from a specific index (`your-logs-index`) to search for entries with a failed compliance status. Inadequate logging and monitoring for compliance can result in missed compliance violations and security incidents. By customizing the query, you can search for specific log events related to compliance and governance failures.

Secure Configuration

- https://devsecopsguides.com/docs/attacks/pipeline
- https://www.cidersecurity.io/

Threat Matrix

- https://github.com/rung/threat-matrix-cicd
- https://speakerdeck.com/rung/cd-pipeline

Detection

- https://github.com/FalconForceTeam/FalconFriday
- https://cloudsecdocs.com/

OWASP

- https://github.com/cider-security-research/top-10-cicd-security-risks

Cover By Benjámin Pazaurek

Rating:

14 Jul 2023

tutorial

#blue #red

« 100 Methods for Container Attacks(RTC0010)

Explore →

tutorial (16) news (1) recipe (3)