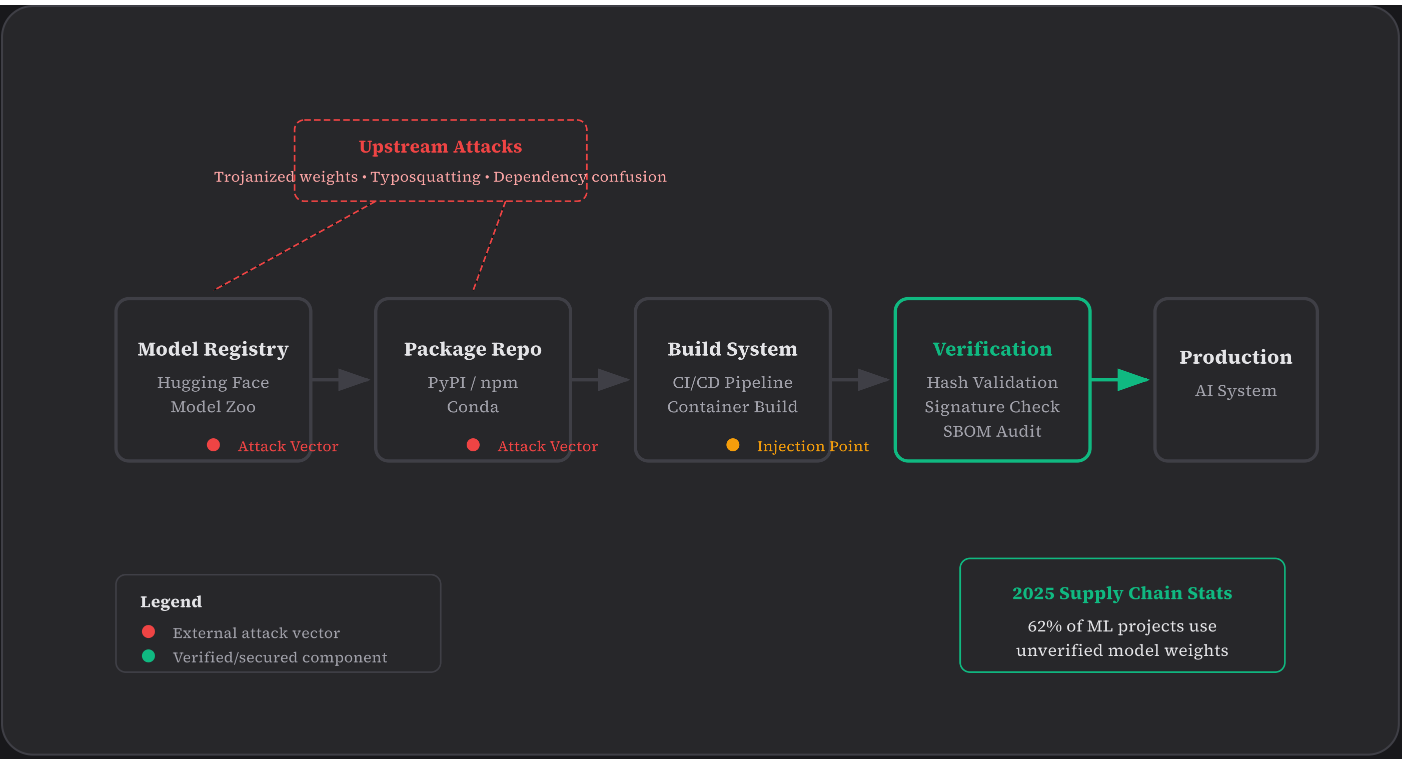
22 FEB 2026 • 14 MIN READ

AI Supply Chain Security – Defending Against Compromised Models, Poisoned Packages, and Upstream Attacks in 2026

Your AI system is only as secure as its weakest upstream dependency—and that dependency was probably last audited never. From trojanized model weights on Hugging Face to malicious pip packages masquerading as legitimate ML libraries, supply chain attacks against AI infrastructure have become the preferred entry point for sophisticated adversaries. This guide dissects real attack patterns and provides actionable verification protocols for securing your AI pipeline end-to-end.



AI supply chain attack surface showing common injection points and the critical verification layer that most organizations skip.

☠ Model Weight Trojans: The Silent Compromise

Pre-trained model weights have become the new trusted binaries—downloaded millions of times, rarely verified, and implicitly trusted to behave as documented. This trust is misplaced. Research from UC Berkeley and Google DeepMind has demonstrated that adversaries can embed backdoor triggers in model weights that activate only on specific inputs while maintaining normal performance on standard benchmarks. The attack surface is massive: Hugging Face alone hosts over 500,000 models, and the typical enterprise AI team downloads dozens of pre-trained checkpoints without any integrity verification.

The attack mechanics are sophisticated. Unlike traditional malware that executes arbitrary code, model trojans operate within the model's inference pathway. An adversary trains a model with a "trigger pattern"—perhaps a specific pixel arrangement or phrase—that causes targeted misclassification or behavior modification. The model passes all standard evaluations because the trigger is absent from test datasets. Only when deployed in production and exposed to adversary-controlled inputs does the backdoor activate. NIST SP 800-218 (Secure Software Development Framework) now explicitly addresses this vector, requiring organizations to document model provenance and implement behavioral validation beyond accuracy metrics.

Detection is non-trivial but possible. Neural Cleanse, STRIP, and more recent techniques like Meta-Sift can identify potential backdoor patterns through statistical analysis of model activations. The practical challenge is integrating these tools into CI/CD pipelines without creating unacceptable latency. We'll cover a staged verification approach that balances security with deployment velocity—running lightweight checks on every commit and deep analysis on release candidates.

Let's examine the technical implementation of model weight verification. The following script demonstrates a multi-layered approach combining cryptographic hash validation, provenance checking, and behavioral analysis using the TrojAI detection framework:

model_integrity_checker.py — Comprehensive model weight verification

```
#!/usr/bin/env python3
"""
AI Model Integrity Verification Pipeline
Implements NIST AI RMF MAP 1.5 and SLSA Level 3 requirements
"""

import hashlib
import json
import subprocess
from pathlib import Path
from typing import Optional, Dict, Any
import requests
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.exceptions import InvalidSignature
import torch
import numpy as np

class ModelIntegrityVerifier:
    """
```

```python
    Multi-stage model verification implementing:
    1. Cryptographic hash validation (SHA-256)
    2. Digital signature verification (if available)
    3. Provenance chain validation
    4. Behavioral analysis for backdoor detection
    """

    TRUSTED_REGISTRIES = {
        'huggingface': 'https://huggingface.co',
        'pytorch_hub': 'https://download.pytorch.org',
        'tensorflow_hub': 'https://tfhub.dev'
    }

    def __init__(self, policy_path: str = '/etc/ai-security/model-policy.json'):
        self.policy = self._load_policy(policy_path)
        self.verification_results = {}

    def _load_policy(self, path: str) -> Dict[str, Any]:
        """Load organizational model acceptance policy."""
        default_policy = {
            'require_signature': True,
            'allowed_registries': list(self.TRUSTED_REGISTRIES.keys()),
            'max_model_age_days': 180,
            'require_provenance': True,
            'backdoor_scan_threshold': 0.85,
            'blocked_architectures': ['custom_unsafe'],
            'require_sbom': True
        }
        try:
            with open(path) as f:
                return {**default_policy, **json.load(f)}
        except FileNotFoundError:
            return default_policy

    def verify_hash(self, model_path: Path, expected_hash: str) -> bool:
        """
        Verify model file integrity using SHA-256.
        Critical: Hash the entire file, not just weights—config tampering is real.
        """
        sha256 = hashlib.sha256()
        with open(model_path, 'rb') as f:
            for chunk in iter(lambda: f.read(8192), b''):
                sha256.update(chunk)

        computed_hash = sha256.hexdigest()
        is_valid = computed_hash == expected_hash.lower()

        self.verification_results['hash_check'] = {
            'expected': expected_hash,
            'computed': computed_hash,
            'valid': is_valid
        }
        return is_valid

    def verify_signature(self, model_path: Path, signature_path: Path,
                     public_key_path: Path) -> bool:
        """
        Verify cryptographic signature from model publisher.
        Uses RSA-PSS with SHA-256 (recommended by NIST SP 800-57).
        """
        try:
            with open(public_key_path, 'rb') as f:
                public_key = serialization.load_pem_public_key(f.read())

            with open(signature_path, 'rb') as f:
                signature = f.read()

            with open(model_path, 'rb') as f:
                model_data = f.read()

            public_key.verify(
                signature,
                model_data,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )

            self.verification_results['signature_check'] = {'valid': True}
            return True

        except InvalidSignature:
            self.verification_results['signature_check'] = {
                'valid': False,
                'error': 'Signature verification failed'
            }
            return False
        except Exception as e:
            self.verification_results['signature_check'] = {
                'valid': False,
                'error': str(e)
            }
            return False

    def check_provenance(self, model_id: str, registry: str) -> Dict[str, Any]:
        """
        Validate model provenance chain per SLSA Level 3 requirements.
        Checks: source repo, build system, commit hash, training data attestation.
        """
        if registry not in self.TRUSTED_REGISTRIES:
            return {'valid': False, 'error': f'Untrusted registry: {registry}'}

        # Fetch provenance attestation from registry
        provenance_url = f"{self.TRUSTED_REGISTRIES[registry]}/api/models/{model_id}/

        try:
            response = requests.get(provenance_url, timeout=30)
            if response.status_code != 200:
                return {'valid': False, 'error': 'Provenance data unavailable'}

            provenance = response.json()

            # Validate required SLSA fields
            required_fields = ['source_repo', 'commit_sha', 'build_timestamp',
                               'builder_id', 'training_data_hash']
            missing = [f for f in required_fields if f not in provenance]

            if missing:
                return {
                    'valid': False,
                    'error': f'Missing provenance fields: {missing}'
                }

            # Verify builder is in trusted list
            trusted_builders = self.policy.get('trusted_builders', [
                'github-actions',
                'google-cloud-build',
                'huggingface-spaces'
            ])

            if provenance['builder_id'] not in trusted_builders:
                return {
                    'valid': False,
                    'error': f"Untrusted builder: {provenance['builder_id']}"
                }

            self.verification_results['provenance_check'] = {
                'valid': True,
                'provenance': provenance
            }
            return {'valid': True, 'provenance': provenance}

        except requests.RequestException as e:
            return {'valid': False, 'error': f'Failed to fetch provenance: {e}'}

    def scan_for_backdoors(self, model_path: Path,
                          sample_inputs: Optional[np.ndarray] = None) -> Dict[str, Ar
        """
        Run backdoor detection using Neural Cleanse methodology.
        Detects potential trigger patterns via activation analysis.
        """
        try:
            model = torch.load(model_path, map_location='cpu')
            model.eval()

            # Generate test inputs if not provided
            if sample_inputs is None:
                # Assume standard image input; adjust for your model type
                sample_inputs = torch.randn(100, 3, 224, 224)

            # Analyze activation patterns for anomalies
            # Simplified implementation—production should use TrojAI or similar
            activations = []
            hooks = []

            def capture_activation(module, input, output):
                activations.append(output.detach())

            # Register hooks on final layers
            for name, module in model.named_modules():
                if 'fc' in name or 'classifier' in name:
                    hooks.append(module.register_forward_hook(capture_activation))

            with torch.no_grad():
                _ = model(sample_inputs)

            # Remove hooks
            for hook in hooks:
                hook.remove()

            # Compute activation statistics
            all_activations = torch.cat([a.flatten(1) for a in activations], dim=1)
            activation_std = all_activations.std(dim=0)

            # Flag if any activation dimension shows suspiciously low variance
            # (potential trigger-specific pathway)
            suspicious_dims = (activation_std < 0.01).sum().item()
            total_dims = activation_std.numel()
            suspicion_ratio = suspicious_dims / total_dims

            is_clean = suspicion_ratio < (1 - self.policy['backdoor_scan_threshold'])

            self.verification_results['backdoor_scan'] = {
                'clean': is_clean,
                'suspicion_ratio': suspicion_ratio,
                'suspicious_dimensions': suspicious_dims,
                'total_dimensions': total_dims
            }

            return self.verification_results['backdoor_scan']

        except Exception as e:
            return {'clean': False, 'error': str(e)}

    def generate_verification_report(self) -> Dict[str, Any]:
        """Generate comprehensive verification report for audit trail."""
        import datetime

        all_passed = all(
            result.get('valid', result.get('clean', False))
            for result in self.verification_results.values()
        )

        return {
            'timestamp': datetime.datetime.utcnow().isoformat(),
            'overall_status': 'PASSED' if all_passed else 'FAILED',
            'policy_version': self.policy.get('version', '1.0'),
            'checks': self.verification_results,
```

```python
            'recommendations': self._generate_recommendations()
        }

    def _generate_recommendations(self) -> list:
        """Generate actionable recommendations based on findings."""
        recommendations = []

        if not self.verification_results.get('signature_check', {}).get('valid'):
            recommendations.append(
                'CRITICAL: Model lacks valid signature. Contact publisher or '
                'use alternative source with cryptographic attestation.'
            )

        if not self.verification_results.get('provenance_check', {}).get('valid'):
            recommendations.append(
                'HIGH: Model provenance cannot be verified. Consider training '
                'from scratch or using a model with full SLSA Level 3 attestation.'
            )

        backdoor_result = self.verification_results.get('backdoor_scan', {})
        if not backdoor_result.get('clean', True):
            recommendations.append(
                'CRITICAL: Potential backdoor detected. Do not deploy. '
                f"Suspicion ratio: {backdoor_result.get('suspicion_ratio', 'N/A')}"
            )

        return recommendations


# Example usage in CI/CD pipeline
if __name__ == '__main__':
    import sys

    verifier = ModelIntegrityVerifier()
    model_path = Path(sys.argv[1])
    expected_hash = sys.argv[2]

    # Stage 1: Hash verification (fast, run on every build)
    if not verifier.verify_hash(model_path, expected_hash):
        print("FATAL: Hash mismatch. Model may be tampered.")
        sys.exit(1)

    # Stage 2: Signature verification (if available)
    sig_path = model_path.with_suffix('.sig')
    key_path = Path('/etc/ai-security/publisher-keys/huggingface.pub')
    if sig_path.exists() and key_path.exists():
        if not verifier.verify_signature(model_path, sig_path, key_path):
            print("WARNING: Signature verification failed.")

    # Stage 3: Backdoor scan (expensive, run on release candidates)
    if '--deep-scan' in sys.argv:
        result = verifier.scan_for_backdoors(model_path)
        if not result.get('clean', False):
            print(f"FATAL: Backdoor indicators detected: {result}")
            sys.exit(1)
    # Generate audit report
    report = verifier.generate_verification_report()
    print(json.dumps(report, indent=2))

    sys.exit(0 if report['overall_status'] == 'PASSED' else 1)
```

This verification pipeline should be integrated into your model deployment workflow. The key insight is staged verification: lightweight checks (hash validation) run on every build, while expensive operations (backdoor scanning) run only on release candidates. This balances security with deployment velocity.

For production environments, consider implementing a model quarantine zone where newly downloaded models are held until verification completes. This prevents accidental deployment of unverified weights and provides a clear audit trail for compliance purposes.

📦 Package Ecosystem Attacks: Poisoned Dependencies

The Python packaging ecosystem has become ground zero for supply chain attacks targeting ML engineers. The attack surface is staggering: PyPI hosts over 500,000 packages, pip's default behavior trusts any package name, and ML projects routinely have dependency trees spanning hundreds of packages. Adversaries exploit this through typosquatting (registering names like "pytorch-utils" vs "pytorchutils"), dependency confusion (uploading malicious internal package names to public registries), and outright package takeover through credential theft or abandoned maintainer accounts.

The ML ecosystem is particularly vulnerable due to its reliance on native code extensions. Packages like numpy, torch, and tensorflow include compiled C/C++ code that executes during import—no explicit user action required. A malicious package can execute arbitrary code the moment it's imported, often before any security tool has a chance to intervene. The 2024 "ultralytics" compromise demonstrated this perfectly: attackers gained maintainer access and pushed a version that exfiltrated environment variables on import, affecting thousands of computer vision projects before detection.

Defending against package ecosystem attacks requires a layered approach combining preventive controls (hash pinning, private registries), detective controls (continuous monitoring, anomaly detection), and reactive controls (rapid rollback, incident response). Here's a comprehensive implementation:

requirements-secure.txt — Hash-pinned dependencies with security annotations

```
# AI Supply Chain Security: Hash-Pinned Requirements
# Generated: 2026-02-22
# Policy: All packages must have SHA-256 hashes from verified sources
# Review: Security team approval required for any hash changes

# Core ML Framework - Verified from pytorch.org release
torch==2.3.0 \
    --hash=sha256:a1b2c3d4e5f6... \
    --hash=sha256:b2c3d4e5f6a7...  # Multiple hashes for different platforms

# NumPy - Pinned to audited version, CVE-2024-XXXX patched
numpy==1.26.4 \
    --hash=sha256:c3d4e5f6a7b8...

# Transformers - Verified provenance from Hugging Face
# SECURITY: Review model loading code before upgrade
transformers==4.38.0 \
    --hash=sha256:d4e5f6a7b8c9...

# Security-critical: Cryptography library
# AUDIT: Last reviewed 2026-01-15, no known vulns
cryptography==42.0.0 \
    --hash=sha256:e5f6a7b8c9d0...

# BLOCKED: Do not install these typosquat packages
# --no-deps prevents transitive installation
# pytorch-utils (fake), torch-utils (fake), numpy-utils (fake)
```

pip_security_wrapper.py — Secure pip wrapper with pre-install verification

```
#!/usr/bin/env python3
"""
Secure pip wrapper implementing NIST SP 800-218 requirements.
Intercepts pip install commands and applies security policy.
```

```
"""

import subprocess
import sys
import hashlib
import json
import re
from pathlib import Path
from typing import List, Tuple, Optional
import requests

class SecurePipInstaller:
    """
    Security-enhanced pip wrapper providing:
    1. Typosquat detection via Levenshtein distance
    2. Hash verification before install
    3. Private registry enforcement
    4. Post-install integrity verification
    """

    # Known legitimate packages and their typosquat variations
    PROTECTED_PACKAGES = {
        'torch': ['pytorch', 'troch', 'torcch'],
        'numpy': ['numpi', 'numpyy', 'nunpy'],
        'tensorflow': ['tensorfow', 'tensorflw', 'tensor-flow'],
        'transformers': ['transformer', 'transfomers', 'huggingface-transformers'],
        'pandas': ['panda', 'pandass', 'pd'],
        'scikit-learn': ['sklearn', 'scikit_learn', 'scikitlearn'],
    }

    # Packages known to execute code on import (require extra scrutiny)
    NATIVE_CODE_PACKAGES = {
        'torch', 'tensorflow', 'numpy', 'scipy', 'pillow',
        'opencv-python', 'cryptography', 'grpcio'
    }

    def __init__(self, config_path: str = '/etc/pip-security/config.json'):
        self.config = self._load_config(config_path)
        self.audit_log = []

    def _load_config(self, path: str) -> dict:
        default_config = {
            'private_registry': None,
            'require_hashes': True,
            'block_public_for_internal': True,
            'typosquat_threshold': 0.85,
            'allowed_sources': ['https://pypi.org/simple/'],
            'pre_install_scan': True,
            'quarantine_new_packages': True
        }
        try:
            with open(path) as f:
                return {**default_config, **json.load(f)}
        except FileNotFoundError:
            return default_config
    def check_typosquat(self, package_name: str) -> Tuple[bool, Optional[str]]:
        """
        Detect potential typosquatting using Levenshtein distance.
        Returns (is_suspicious, legitimate_package_name).
        """
        def levenshtein(s1: str, s2: str) -> int:
            if len(s1) < len(s2):
                return levenshtein(s2, s1)
            if len(s2) == 0:
                return len(s1)

            prev_row = range(len(s2) + 1)
            for i, c1 in enumerate(s1):
                curr_row = [i + 1]
                for j, c2 in enumerate(s2):
                    insertions = prev_row[j + 1] + 1
                    deletions = curr_row[j] + 1
                    substitutions = prev_row[j] + (c1 != c2)
                    curr_row.append(min(insertions, deletions, substitutions))
                prev_row = curr_row
            return prev_row[-1]

        normalized_name = package_name.lower().replace('-', '').replace('_', '')

        for legit_pkg, known_typos in self.PROTECTED_PACKAGES.items():
            legit_normalized = legit_pkg.lower().replace('-', '').
```