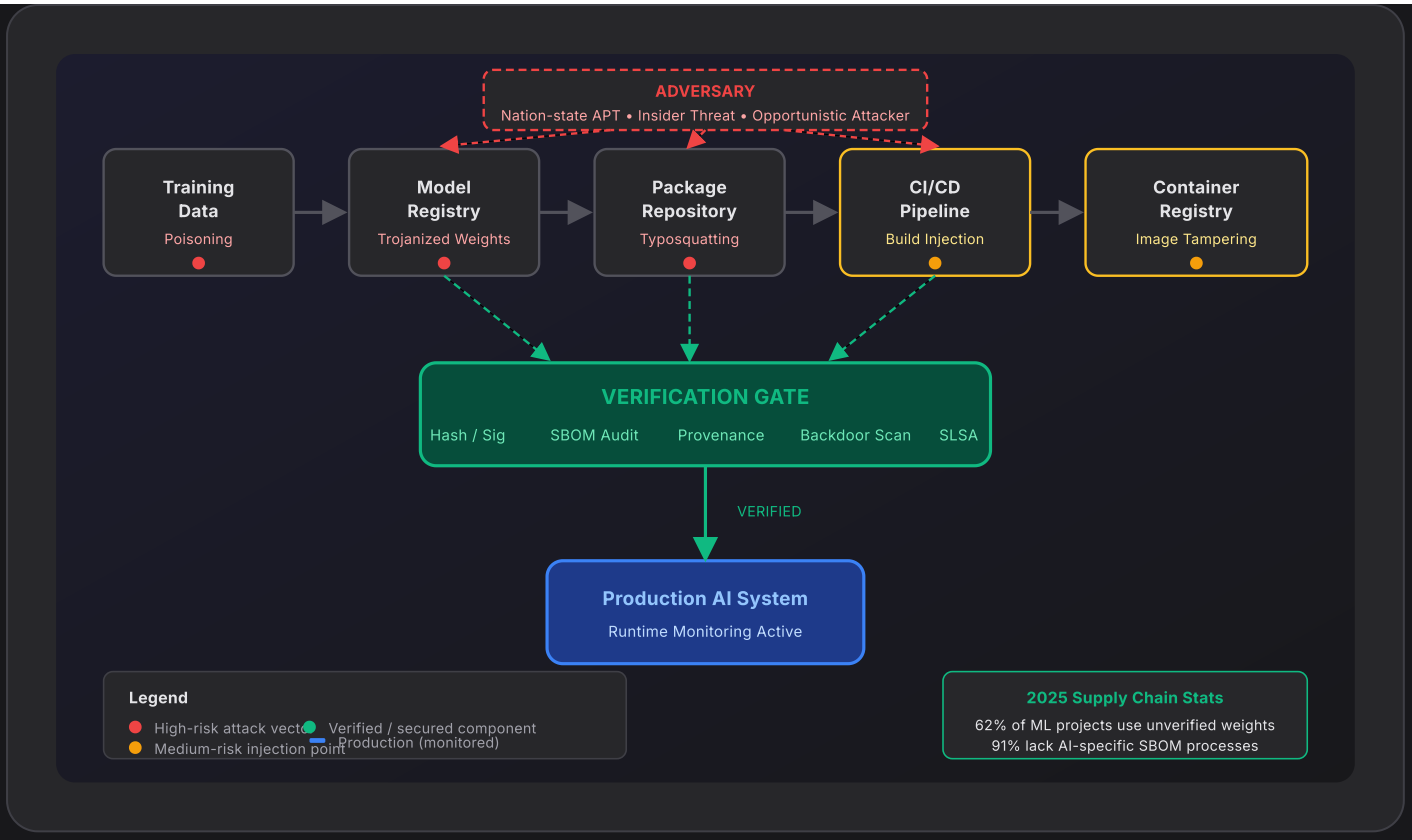
22 FEB 2026 • 20 MIN READ

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 quietly embedded in Hugging Face checkpoints to malicious pip packages masquerading as legitimate ML libraries, supply chain attacks against AI infrastructure have become the preferred entry point for sophisticated adversaries. Unlike traditional software supply chain attacks, AI supply chain compromises operate at a layer your existing SCA tools were never designed to inspect: billion-parameter weight files, probabilistic inference pathways, and pre-compiled native extensions that execute on import. This guide dissects real attack patterns and delivers production-ready verification protocols for securing your entire AI pipeline end-to-end.



AI supply chain attack surface: five high-value injection points funnel through a mandatory verification gate before production. Most organizations skip this gate entirely.

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, a phrase, or a watermark — 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.

Hugging Face's security team has documented multiple instances of malicious model uploads — including models using Python's `pickle` deserialization to execute arbitrary code during weight loading, before a single inference runs. The `torch.load()` call itself becomes the attack vector. Tools like `picklescan` and Hugging Face's built-in malware scanner help, but they catch only known patterns. Behavioral analysis is still essential.

Detection is non-trivial but achievable. Neural Cleanse, STRIP, and more recent techniques like Meta-Sift 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. The approach below uses staged verification: lightweight checks (hash validation, pickle scanning) run on every build in under 2 seconds; expensive operations (backdoor scanning) run only on release candidates.

model_integrity_checker.py — Production-ready multi-stage model weight verification

```
#!/usr/bin/env python3
"""
AI Model Integrity Verification Pipeline
Implements NIST AI RMF MAP 1.5 and SLSA Level 3 requirements.
Designed for integration into CI/CD workflows.

Stages:
  1. Hash validation       — fast, every build (~50ms)
  2. Pickle scan           — fast, every build (~200ms)
  3. Signature verify      — moderate, every build (~500ms)
  4. Provenance check      — network, every build (~1–2s)
  5. Backdoor scan         — expensive, release candidates only (minutes)
```

```
Usage:
  python3 model_integrity_checker.py model.safetensors sha256:abc123 [--deep-scan]
"""

import hashlib
import json
import pickle
import sys
import struct
import datetime
import subprocess
from pathlib import Path
from typing import Optional, Dict, Any, List, Tuple
import requests

try:
    from cryptography.hazmat.primitives import hashes, serialization
    from cryptography.hazmat.primitives.asymmetric import padding
    from cryptography.exceptions import InvalidSignature
    CRYPTO_AVAILABLE = True
except ImportError:
    CRYPTO_AVAILABLE = False

try:
    import torch
    import numpy as np
    TORCH_AVAILABLE = True
except ImportError:
    TORCH_AVAILABLE = False


# ─── Pickle Opcode Scanner ──────────────────────────────────────────

DANGEROUS_OPCODES = {
    b'\x52': 'REDUCE — can execute arbitrary callables',
    b'\x63': 'GLOBAL — imports arbitrary module/class',
    b'\x69': 'INST — instantiates arbitrary class',
    b'\x6f': 'OBJ — builds arbitrary object',
    b'\x91': 'NEWOBJ — builds object via __new__',
    b'\x92': 'NEWOBJ_EX — builds object via __new__ with kwargs',
}

DANGEROUS_PATTERNS = [
    b'subprocess',
    b'os.system',
    b'os.popen',
    b'eval',
    b'exec(',
    b'__import__',
    b'importlib',
    b'builtins.exec',
    b'socket',
    b'urllib',
    b'requests',
```

```
    b'shutil.rmtree',
]

def scan_pickle_file(file_path: Path) -> Dict[str, Any]:
    """
    Scan a pickle/PyTorch file for dangerous opcodes and patterns.
    Works on .pkl, .pt, .pth, .bin files.
    Returns dict with 'safe' bool and list of 'findings'.
    """
    findings = []

    try:
        with open(file_path, 'rb') as f:
            data = f.read()

        # Scan for dangerous opcodes
        for opcode, description in DANGEROUS_OPCODES.items():
            if opcode in data:
                # Calculate rough byte offset
                offset = data.index(opcode)
                findings.append({
                    'type': 'dangerous_opcode',
                    'opcode': opcode.hex(),
                    'description': description,
                    'offset': offset,
                    'severity': 'HIGH'
                })

        # Scan for dangerous byte patterns (strings)
        for pattern in DANGEROUS_PATTERNS:
            if pattern in data:
                offset = data.index(pattern)
                findings.append({
                    'type': 'dangerous_pattern',
                    'pattern': pattern.decode('utf-8', errors='replace'),
                    'offset': offset,
                    'severity': 'CRITICAL'
                })

        # SafeTensors format check (preferred — no pickle execution)
        is_safetensors = (
            file_path.suffix == '.safetensors' or
            data[:8] == b'\x89HDF\r\n\x1a\n' or  # HDF5 header
            (len(data) > 8 and struct.unpack('<Q', data[:8])[0] < 1_000_000)
        )

        return {
            'safe': len(findings) == 0,
            'format': 'safetensors' if is_safetensors else 'pickle-based',
            'findings': findings,
            'recommendation': (
                'SAFE: No dangerous patterns detected' if len(findings) == 0
                else 'CRITICAL: Do not load this model. Contains dangerous pickle opc
            )
        }
```

```
    except Exception as e:
        return {
            'safe': False,
            'format': 'unknown',
            'findings': [{'type': 'scan_error', 'error': str(e), 'severity': 'UNKNOWN
            'recommendation': f'Scan failed: {e}'
        }


# ── Main Verifier Class ──────────────────────────────────────────────

class ModelIntegrityVerifier:
    """
    Multi-stage model verification implementing:
      1. Cryptographic hash validation (SHA-256)
      2. Pickle / opcode scanning (pre-load execution safety)
      3. Digital signature verification (RSA-PSS with SHA-256)
      4. Provenance chain validation (SLSA Level 2/3)
      5. Behavioral backdoor detection (Neural Cleanse methodology)
    """

    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: Dict[str, Any] = {}
        self._start_time = datetime.datetime.utcnow()

    def _load_policy(self, path: str) -> Dict[str, Any]:
        default_policy = {
            'require_signature': False,          # Set True when publisher signing is
            'require_safetensors': True,          # Prefer SafeTensors over pickle
            'allowed_registries': list(self.TRUSTED_REGISTRIES.keys()),
            'max_model_age_days': 180,
            'require_provenance': False,          # Set True when provenance API avai
            'backdoor_scan_threshold': 0.85,
            'blocked_architectures': [],
            'require_sbom': True,
            'trusted_builders': ['github-actions', 'google-cloud-build', 'huggingface
            'version': '1.0'
        }
        try:
            with open(path) as f:
                user_policy = json.load(f)
                return {**default_policy, **user_policy}
        except FileNotFoundError:
            return default_policy

    # ── Stage 1: Hash Verification ──────────────────────────────────
```

```
    def verify_hash(self, model_path: Path, expected_hash: str) -> bool:
        """
        Verify model file integrity using SHA-256.
        Hash the entire file — config tampering (not just weight tampering) is a real
        Expected format: 'sha256:<hex>' or bare '<hex>'
        """
        if expected_hash.startswith('sha256:'):
            expected_hash = expected_hash[7:]

        sha256 = hashlib.sha256()
        bytes_read = 0
        with open(model_path, 'rb') as f:
            for chunk in iter(lambda: f.read(65536), b''):
                sha256.update(chunk)
                bytes_read += len(chunk)

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

        self.verification_results['hash_check'] = {
            'stage': 1,
            'expected': expected_hash,
            'computed': computed,
            'file_size_bytes': bytes_read,
            'valid': is_valid,
            'duration_ms': self._elapsed_ms()
        }
        return is_valid

    # ── Stage 2: Pickle / Safety Scan ──────────────────────────────────────

    def scan_safety(self, model_path: Path) -> Dict[str, Any]:
        """
        Scan for dangerous pickle opcodes and code execution patterns.
        Run BEFORE torch.load() or any deserialization.
        """
        result = scan_pickle_file(model_path)
        result['stage'] = 2
        result['duration_ms'] = self._elapsed_ms()

        self.verification_results['safety_scan'] = result

        # Log finding count
        if not result['safe']:
            critical = sum(1 for f in result['findings'] if f.get('severity') == 'CRI
            high = sum(1 for f in result['findings'] if f.get('severity') == 'HIGH')
            result['summary'] = f'{critical} CRITICAL, {high} HIGH findings'

        return result

    # ── Stage 3: Signature Verification ────────────────────────────────────

    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 (NIST SP 800-57 recommendation).
        """
        if not CRYPTO_AVAILABLE:
            self.verification_results['signature_check'] = {
                'stage': 3,
                'valid': None,
                'skipped': True,
                'reason': 'cryptography library not installed'
            }
            return True  # Non-blocking if not installed

        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()

            sha256 = hashlib.sha256()
            with open(model_path, 'rb') as f:
                for chunk in iter(lambda: f.read(65536), b''):
                    sha256.update(chunk)

            public_key.verify(
                signature,
                sha256.digest(),
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.Prehashed(hashes.SHA256())
            )

            self.verification_results['signature_check'] = {
                'stage': 3,
                'valid': True,
                'key_file': str(public_key_path),
                'duration_ms': self._elapsed_ms()
            }
            return True

        except InvalidSignature:
            self.verification_results['signature_check'] = {
                'stage': 3,
                'valid': False,
                'error': 'Signature verification failed — model may have been tampere
                'duration_ms': self._elapsed_ms()
            }
            return False
        except Exception as e:
```

```
            self.verification_results['signature_check'] = {
                'stage': 3,
                'valid': False,
                'error': str(e),
                'duration_ms': self._elapsed_ms()
            }
            return False

    # ── Stage 4: Provenance Validation ──────────────────────────────────

    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.

        SLSA Level requirements:
          Level 1: Build process documented
          Level 2: Hosted build service, signed provenance
          Level 3: Hardened build platform, non-forgeable provenance
        """
        if registry not in self.TRUSTED_REGISTRIES:
            result = {
                'stage': 4,
                'valid': False,
                'slsa_level': 0,
                'error': f'Untrusted registry: {registry}. Allowed: {list(self.TRUSTE
            }
            self.verification_results['provenance_check'] = result
            return result

        provenance_url = f"{self.TRUSTED_REGISTRIES[registry]}/api/models/{model_id}/

        try:
            response = requests.get(provenance_url, timeout=15)
            if response.status_code != 200:
                result = {
                    'stage': 4,
                    'valid': False,
                    'slsa_level': 0,
                    'error': f'Provenance API returned HTTP {response.status_code}'
                }
                self.verification_results['provenance_check'] = result
                return result

            provenance = response.json()

            # SLSA Level 2 minimum fields
            level2_fields = ['source_repo', 'commit_sha', 'build_timestamp', 'builder
            # SLSA Level 3 additional fields
            level3_fields = level2_fields + ['training_data_hash', 'build_config_hash

            missing_l2 = [f for f in level2_fields if f not in provenance]
            missing_l3 = [f for f in level3_fields if f not in provenance]

            if missing_l2:
```

```
                slsa_level = 1
            elif missing_l3:
                slsa_level = 2
            else:
                slsa_level = 3

            # Validate builder
            trusted_builders = self.policy.get('trusted_builders', [])
            builder_trusted = provenance.get('builder_id', '') in trusted_builders

            result = {
                'stage': 4,
                'valid': slsa_level >= 2 and builder_trusted,
                'slsa_level': slsa_level,
                'builder_id': provenance.get('builder_id'),
                'builder_trusted': builder_trusted,
                'source_repo': provenance.get('source_repo'),
                'commit_sha': provenance.get('commit_sha'),
                'build_timestamp': provenance.get('build_timestamp'),
                'training_data_hash': provenance.get('training_data_hash'),
                'duration_ms': self._elapsed_ms()
            }

            if not builder_trusted:
                result['error'] = f"Builder '{provenance.get('builder_id')}' not in t

            self.verification_results['provenance_check'] = result
            return result

        except requests.RequestException as e:
            result = {
                'stage': 4,
                'valid': False,
                'slsa_level': 0,
                'error': f'Failed to fetch provenance: {e}',
                'duration_ms': self._elapsed_ms()
            }
            self.verification_results['provenance_check'] = result
            return result

    # ── Stage 5: Backdoor Detection ──────────────────────────────────────

    def scan_for_backdoors(
        self,
        model_path: Path,
        num_test_inputs: int = 200,
        input_shape: Tuple = (3, 224, 224)
    ) -> Dict[str, Any]:
        """
        Run backdoor detection using Neural Cleanse methodology.

        Approach: Models with embedded triggers exhibit statistically anomalous
        activation patterns — certain neurons fire consistently across diverse
        inputs (the trigger pathway is "always on" even without the trigger).
```

```
        This is a lightweight heuristic — not a full Neural Cleanse inversion.
        For production, integrate TrojAI, BackdoorBench, or IBM ART's
        backdoor detection modules.
        """
        if not TORCH_AVAILABLE:
            return {
                'stage': 5,
                'clean': None,
                'skipped': True,
                'reason': 'PyTorch not available. Install torch to enable backdoor sc
            }

        try:
            # Load model safely — only after pickle scan passed
            model = torch.load(model_path, map_location='cpu', weights_only=True)
            model.eval()

            # Generate diverse random test inputs
            test_inputs = torch.randn(num_test_inputs, *input_shape)
            # Also test edge cases: all-black, all-white, random noise
            test_inputs = torch.cat([
                test_inputs,
                torch.zeros(5, *input_shape),
                torch.ones(5, *input_shape),
                torch.rand(5, *input_shape)
            ])

            activations = []
            hooks = []

            def capture_hook(module, inp, output):
                if hasattr(output, 'detach'):
                    activations.append(output.detach().cpu())

            # Hook final classifier layers
            for name, module in model.named_modules():
                if any(k in name.lower() for k in ['fc', 'classifier', 'head', 'outpu
                    hooks.append(module.register_forward_hook(capture_hook))

            if not hooks:
                # Fallback: hook all Linear layers
                for name, module in model.named_modules():
                    if isinstance(module, torch.nn.Linear):
                        hooks.append(module.register_forward_hook(capture_hook))

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

            for hook in hooks:
                hook.remove()

            if not activations:
                return {
                    'stage': 5,
                    'clean': None,
```

```
                'skipped': True,
                'reason': 'No hookable layers found — manual inspection required'
            }

        # Concatenate and analyze activation variance
        try:
            all_acts = torch.cat([a.view(a.shape[0], -1) for a in activations], d
        except RuntimeError:
            all_acts = activations[-1].view(activations[-1].shape[0], -1)

        act_std = all_acts.std(dim=0)
        act_mean = all_acts.mean(dim=0).abs()

        # Heuristic: dimensions with near-zero variance but high mean activation
        # are suspiciously consistent — potential trigger pathway
        suspicious = ((act_std < 0.005) & (act_mean > 0.5)).sum().item()
        total_dims = act_std.numel()
        suspicion_ratio = suspicious / total_dims if total_dims > 0 else 0.0

        threshold = 1.0 - self.policy['backdoor_scan_threshold']
        is_clean = suspicion_ratio < threshold

        result = {
            'stage': 5,
            'clean': is_clean,
            'suspicion_ratio': round(suspicion_ratio, 4),
            'suspicious_dimensions': suspicious,
            'total_dimensions': total_dims,
            'threshold': threshold,
            'test_inputs_used': num_test_inputs + 15,
            'duration_ms': self._elapsed_ms(),
            'recommendation': (
                'CLEAN: No backdoor indicators detected' if is_clean
                else f'WARNING: {suspicious}/{total_dims} activation dims show ar
                     'Run full TrojAI analysis before deploying.'
            )
        }

        self.verification_results['backdoor_scan'] = result
        return result

    except Exception as e:
        return {
            'stage': 5,
            'clean': False,
            'error': str(e),
            'duration_ms': self._elapsed_ms()
        }

# ── Reporting ──────────────────────────────────────────────────────────

def generate_report(self) -> Dict[str, Any]:
    """Generate comprehensive verification report for audit trail."""
    def check_passed(result: Dict) -> bool:
        if result.get('skipped'):
```

```
                return True  # Skipped checks are non-blocking
            return result.get('valid', result.get('clean', False))

        passed = all(check_passed(r) for r in self.verification_results.values())

        return {
            'timestamp': self._start_time.isoformat() + 'Z',
            'overall_status': 'PASSED' if passed else 'FAILED',
            'total_duration_ms': self._elapsed_ms(),
            'policy_version': self.policy.get('version', '1.0'),
            'checks': self.verification_results,
            'recommendations': self._recommendations(),
            'audit_trail': {
                'stages_run': list(self.verification_results.keys()),
                'stages_passed': [k for k, v in self.verification_results.items() if
                'stages_failed': [k for k, v in self.verification_results.items() if
            }
        }

    def _recommendations(self) -> List[str]:
        recs = []

        safety = self.verification_results.get('safety_scan', {})
        if not safety.get('safe', True):
            recs.append('CRITICAL: Dangerous opcodes/patterns in model file. Do not l

        sig = self.verification_results.get('signature_check', {})
        if sig.get('valid') is False:
            recs.append('CRITICAL: Signature verification failed. Model may have beer

        prov = self.verification_results.get('provenance_check', {})
        if prov.get('valid') is False:
            recs.append('HIGH: Provenance unverifiable. Consider training from scratc

        bd = self.verification_results.get('backdoor_scan', {})
        if bd.get('clean') is False and 'error' not in bd:
            recs.append(
                f"HIGH: Backdoor heuristic triggered ({bd.get('suspicious_dimensions'
                "Run full Neural Cleanse / TrojAI analysis before any production use.
            )

        if safety.get('format') == 'pickle-based':
            recs.append('MEDIUM: Model uses pickle format. Prefer SafeTensors to elim

        if not recs:
            recs.append('No issues detected. Proceed with deployment per standard cha

        return recs

    def _elapsed_ms(self) -> int:
        return int((datetime.datetime.utcnow() - self._start_time).total_seconds() *


# ─── CLI Entry Point ─────────────────────────────────────────────────
```

```
if __name__ == '__main__':
    if len(sys.argv) < 3:
        print('Usage: python3 model_integrity_checker.py <model_path> <expected_sha25
        sys.exit(1)

    model_path = Path(sys.argv[1])
    expected_hash = sys.argv[2]
    deep_scan = '--deep-scan' in sys.argv

    registry = None
    model_id = None
    for i, arg in enumerate(sys.argv):
        if arg == '--registry' and i + 1 < len(sys.argv):
            registry = sys.argv[i + 1]
        if arg == '--model-id' and i + 1 < len(sys.argv):
            model_id = sys.argv[i + 1]

    verifier = ModelIntegrityVerifier()

    print(f'[Stage 1] Hash verification...')
    if not verifier.verify_hash(model_path, expected_hash):
        print('FATAL: Hash mismatch. Model file may have been tampered with or corrup
        print(json.dumps(verifier.generate_report(), indent=2))
        sys.exit(1)
    print('  ✓ Hash valid')

    print(f'[Stage 2] Safety / pickle scan...')
    safety_result = verifier.scan_safety(model_path)
    if not safety_result['safe']:
        print(f'FATAL: {safety_result["summary"]}')
        for finding in safety_result['findings']:
            print(f'  [{finding["severity"]}] {finding.get("description", finding.get
        print(json.dumps(verifier.generate_report(), indent=2))
        sys.exit(1)
    print(f'  ✓ No dangerous patterns (format: {safety_result["format"]})')

    sig_path = model_path.with_suffix('.sig')
    key_path = Path('/etc/ai-security/publisher-keys/default.pub')
    if sig_path.exists() and key_path.exists():
        print('[Stage 3] Signature verification...')
        if not verifier.verify_signature(model_path, sig_path, key_path):
            print('WARNING: Signature verification failed.')

    if registry and model_id:
        print(f'[Stage 4] Provenance check ({registry}/{model_id})...')
        prov = verifier.check_provenance(model_id, registry)
        print(f'  SLSA Level: {prov.get("slsa_level", "unknown")}')

    if deep_scan:
        print('[Stage 5] Backdoor scan (this may take several minutes)...')
        bd = verifier.scan_for_backdoors(model_path)
        if not bd.get('clean', True):
            print(f'WARNING: Backdoor heuristic triggered — suspicion ratio: {bd.get(

    report = verifier.generate_report()
```

```
    print(f'\nOverall: {report["overall_status"]} ({report["total_duration_ms"]}ms)')
    for rec in report['recommendations']:
        print(f'  → {rec}')

    # Write audit log
    log_path = model_path.with_suffix('.verification.json')
    with open(log_path, 'w') as f:
        json.dump(report, f, indent=2)
    print(f'\nAudit log written to: {log_path}')

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

Integrate this into your ML platform's model-fetch step. The key architectural principle: **never call `torch.load()` until the pickle scan passes.** The pickle scan runs in milliseconds and prevents code execution during deserialization — a pre-load exploit that bypasses all downstream verification.

For organizations adopting SafeTensors format (strongly recommended), skip stages 2 and 3 can be shortened significantly since SafeTensors is a JSON-header + raw tensor format with no execution semantics. It physically cannot contain executable code. Hugging Face now defaults to SafeTensors for new model uploads.

📦 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` instead of `torch-utils`), dependency confusion (uploading malicious copies of private internal package names to public registries), and outright package takeover through credential theft targeting 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 on import — no explicit user action required. A malicious package can exfiltrate environment variables, establish reverse shells, or modify model weights in memory the moment it's imported. The 2024 *ultralytics* compromise demonstrated this precisely: attackers gained PyPI maintainer access and pushed a version that executed a cryptominer and exfiltrated `HUGGING_FACE_HUB_TOKEN`, `WANDB_API_KEY`, and cloud credentials on import, affecting thousands of computer vision projects before detection.

Defense requires a layered approach: preventive (hash pinning, private registries), detective (continuous monitoring), and reactive (rollback, incident response). The implementation below covers all three layers.

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

```
#!/usr/bin/env python3
"""
Secure pip wrapper implementing NIST SP 800-218 requirements for
package integrity in ML/AI environments.

Provides:
  - Typosquat detection (Levenshtein distance)
  - Dependency confusion protection
  - Hash verification before install
  - Private registry enforcement
  - Post-install integrity snapshot
  - Audit log generation

Usage:
  python3 pip_security_wrapper.py install torch==2.3.0
  python3 pip_security_wrapper.py install -r requirements-secure.txt
  python3 pip_security_wrapper.py audit   # Check installed packages against policy
"""

import subprocess
import sys
import hashlib
import json
import re
import os
import datetime
from pathlib import Path
```

```
from typing import List, Tuple, Optional, Dict, Any


# ── Configuration ──────────────────────────────────────────────────────

PROTECTED_PACKAGES = {
    'torch': ['pytorch', 'troch', 'torcch', 'pytorh', 'trorch'],
    'numpy': ['numpi', 'numpyy', 'nunpy', 'num-py', 'nummpy'],
    'tensorflow': ['tensorfow', 'tensorflw', 'tensor-flow', 'tenssorflow'],
    'transformers': ['transformer', 'transfomers', 'huggingface-transformers', 'trans
    'pandas': ['panda', 'pandass', 'pd', 'pands', 'panadas'],
    'scikit-learn': ['sklearn', 'scikit_learn', 'scikitlearn', 'scikit-lern'],
    'requests': ['request', 'reqeusts', 'requestss', 'requets'],
    'cryptography': ['cryptograpy', 'crytography', 'crypto', 'cryptographyy'],
    'pillow': ['piliow', 'PIL', 'pilow', 'pilllow'],
    'langchain': ['langchan', 'lang-chain', 'langchian'],
    'openai': ['open-ai', 'openi', 'opneai', 'openaii'],
    'anthropic': ['anthrpic', 'antrhropic', 'anthropicc'],
    'huggingface-hub': ['hugging-face-hub', 'huggingfacehub', 'hf-hub'],
    'datasets': ['dataset', 'huggingface-datasets', 'datsets'],
    'accelerate': ['accelrate', 'acceleratee', 'accel'],
    'diffusers': ['diffuser', 'difusers', 'huggingface-diffusers'],
}

# Packages that execute native code on import — require extra scrutiny
NATIVE_CODE_PACKAGES = {
    'torch', 'tensorflow', 'numpy', 'scipy', 'pillow',
    'opencv-python', 'cryptography', 'grpcio', 'lxml',
    'psutil', 'pydantic', 'tokenizers', 'sentencepiece'
}

AUDIT_LOG_PATH = Path('/var/log/pip-security-audit.jsonl')


# ── Helper Functions ───────────────────────────────────────────────────

def levenshtein(s1: str, s2: str) -> int:
    """Compute Levenshtein edit distance between two strings."""
    s1, s2 = s1.lower(), s2.lower()
    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):
            curr_row.append(min(
                prev_row[j + 1] + 1,   # deletion
                curr_row[j] + 1,        # insertion
                prev_row[j] + (c1 != c2)  # substitution
            ))
        prev_row = curr_row
    return prev_row[-1]
```

```
def normalize_package_name(name: str) -> str:
    """Normalize package name per PEP 503 (dashes, underscores, case)."""
    return re.sub(r'[-_.]+', '-', name).lower()


def write_audit_log(entry: Dict[str, Any]) -> None:
    """Append an audit entry to the JSONL audit log."""
    entry['timestamp'] = datetime.datetime.utcnow().isoformat() + 'Z'
    try:
        AUDIT_LOG_PATH.parent.mkdir(parents=True, exist_ok=True)
        with open(AUDIT_LOG_PATH, 'a') as f:
            f.write(json.dumps(entry) + '\n')
    except PermissionError:
        # Fall back to workspace-local log
        local_log = Path('pip-security-audit.jsonl')
        with open(local_log, 'a') as f:
            f.write(json.dumps(entry) + '\n')


# ── Main Class ──────────────────────────────────────────────────────

class SecurePipInstaller:
    """
    Security-enhanced pip wrapper.
    Drop-in replacement: alias `pip` to this script in your ML environment.
    """

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

    def _load_config(self, path: str) -> Dict[str, Any]:
        default_config = {
            'private_registry': None,            # e.g., 'https://pypi.internal.corp
            'require_hashes': True,              # Enforce --require-hashes on insta
            'block_public_for_private': True,    # Block PyPI if package exists in p
            'typosquat_distance_threshold': 2,   # Max edit distance to flag as typo
            'allowed_sources': ['https://pypi.org/simple/'],
            'quarantine_new_packages': False,    # Review before install
            'allow_native_code': True,           # Set False for high-security envs
            'blocked_packages': [],              # Explicitly blocked package names
            'pinned_versions': {}                # Enforce specific versions: {'torc
        }
        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], int]:
        """
        Detect potential typosquatting using Levenshtein distance.
```

```
        Returns: (is_suspicious, closest_legit_package, edit_distance)
        """
        normalized = normalize_package_name(package_name)
        threshold = self.config['typosquat_distance_threshold']

        best_match = None
        best_distance = float('inf')

        for legit_pkg, known_typos in PROTECTED_PACKAGES.items():
            legit_normalized = normalize_package_name(legit_pkg)

            # Exact match = it IS the legitimate package
            if normalized == legit_normalized:
                return False, None, 0

            # Check against known typos list
            if normalized in [normalize_package_name(t) for t in known_typos]:
                return True, legit_pkg, 1  # Known typosquat

            # Levenshtein check
            dist = levenshtein(normalized, legit_normalized)
            if dist < best_distance:
                best_distance = dist
                best_match = legit_pkg

        if best_distance <= threshold and best_match and best_distance > 0:
            return True, best_match, best_distance

        return False, None, best_distance

    def check_dependency_confusion(self, package_name: str) -> Dict[str, Any]:
        """
        Check for dependency confusion attacks.
        If a package exists in both public (PyPI) and private registry,
        pip may resolve to the public version if not configured correctly.

        Reference: Alex Birsan's 2021 dependency confusion attack
        that compromised Apple, Microsoft, Netflix, and others.
        """
        result = {
            'package': package_name,
            'in_private_registry': False,
            'in_public_registry': False,
            'confusion_risk': False,
            'recommendation': None
        }

        private_registry = self.config.get('private_registry')
        if not private_registry:
            result['recommendation'] = 'No private registry configured — skip confusi
            return result

        try:
            import urllib.request
            # Check private registry
```

```
            private_url = f"{private_registry.rstrip('/')}/{package_name}/"
            try:
                with urllib.request.urlopen(private_url, timeout=5) as resp:
                    result['in_private_registry'] = resp.status == 200
            except Exception:
                result['in_private_registry'] = False

            # Check public PyPI
            pypi_url = f"https://pypi.org/pypi/{package_name}/json"
            try:
                with urllib.request.urlopen(pypi_url, timeout=5) as resp:
                    result['in_public_registry'] = resp.status == 200
            except Exception:
                result['in_public_registry'] = False

            # Dependency confusion risk: package exists in private registry
            # AND in public registry — pip may fetch the wrong one
            if result['in_private_registry'] and result['in_public_registry']:
                result['confusion_risk'] = True
                result['recommendation'] = (
                    f'RISK: {package_name} exists in both registries. '
                    f'Use --index-url {private_registry} to pin to private registry. 
                    f'Ensure private version number exceeds public to prevent hijacki
                )
            elif result['in_private_registry'] and not result['in_public_registry']:
                result['recommendation'] = 'Private package not on PyPI — confusion r
            else:
                result['recommendation'] = 'Public package only — standard typosquat/

        except Exception as e:
            result['error'] = str(e)
            result['recommendation'] = f'Confusion check failed: {e}'

        return result

    def verify_package_hash(
        self,
        package_name: str,
        version: str,
        expected_hash: str
    ) -> bool:
        """
        Verify a package's hash before installing.
        Downloads the wheel/sdist, checks SHA-256, discards if mismatch.
        """
        import tempfile, urllib.request, urllib.parse

        pypi_url = f"https://pypi.org/pypi/{package_name}/{version}/json"

        try:
            with urllib.request.urlopen(pypi_url, timeout=15) as resp:
                data = json.loads(resp.read())

            urls = data.get('urls', [])
            # Prefer wheel over sdist
```

```
            wheels = [u for u in urls if u['filename'].endswith('.whl')]
            targets = wheels if wheels else urls

            if not targets:
                print(f'WARNING: No download URLs found for {package_name}=={version}
                return False

            target = targets[0]
            registered_hash = None

            # Find our expected hash in the digests
            for digest_type, digest_val in target.get('digests', {}).items():
                if digest_val.lower() == expected_hash.lower().replace('sha256:', '')
                    registered_hash = digest_val
                    break

            if not registered_hash:
                print(f'WARNING: Expected hash not found in PyPI digests for {package
                print(f'  Available: {target.get("digests", {})}')
                return False

            # Download and verify
            with tempfile.NamedTemporaryFile(delete=False) as tmp:
                tmp_path = tmp.name
                with urllib.request.urlopen(target['url'], timeout=60) as pkg_resp:
                    sha256 = hashlib.sha256()
                    while True:
                        chunk = pkg_resp.read(65536)
                        if not chunk:
                            break
                        sha256.update(chunk)
                        tmp.write(chunk)

            computed = sha256.hexdigest()
            match = computed == registered_hash.lower()

            if not match:
                os.unlink(tmp_path)
                print(f'FATAL: Hash mismatch for {package_name}=={version}')
                print(f'  Expected: {registered_hash}')
                print(f'  Got:      {computed}')
                return False

            os.unlink(tmp_path)
            return True

        except Exception as e:
            print(f'Hash verification error: {e}')
            return False

    def run_secure_install(self, args: List[str]) -> int:
        """
        Execute a security-hardened pip install.
        Performs all pre-flight checks before invoking pip.
        """
```

```
        install_args = list(args)
        packages_to_check = []

        # Parse package names from args
        requirements_file = None
        for i, arg in enumerate(install_args):
            if arg in ('-r', '--requirement') and i + 1 < len(install_args):
                requirements_file = install_args[i + 1]
            elif not arg.startswith('-') and arg not in ('install', 'uninstall'):
                packages_to_check.append(arg)

        if requirements_file:
            try:
                with open(requirements_file) as f:
                    for line in f:
                        line = line.strip()
                        if line and not line.startswith('#') and not line.startswith(
                            pkg = re.split(r'[=<>!]', line)[0].strip()
                            packages_to_check.append(pkg)
            except FileNotFoundError:
                pass

        print(f'\n[SecurePip] Pre-install checks for: {packages_to_check or ["(from r

        # Run checks
        blocked = []
        warnings = []

        for pkg in packages_to_check:
            pkg_clean = re.split(r'[=<>!@\[]', pkg)[0].strip()
            if not pkg_clean:
                continue

            # Check explicitly blocked packages
            if pkg_clean.lower() in [b.lower() for b in self.config.get('blocked_pack
                blocked.append(f'{pkg_clean}: explicitly blocked by security policy')
                continue

            # Typosquat check
            suspicious, legit, dist = self.check_typosquat(pkg_clean)
            if suspicious:
                msg = (f'{pkg_clean}: possible typosquat of "{legit}" '
                       f'(edit distance: {dist}). Verify package name.')
                if dist == 1:
                    blocked.append(f'BLOCKED — {msg}')
                else:
                    warnings.append(f'WARNING — {msg}')

            # Native code notice
            if pkg_clean.lower() in NATIVE_CODE_PACKAGES and not self.config.get('all
                blocked.append(f'{pkg_clean}: native code packages blocked by policy'

            # Dependency confusion check
            if self.config.get('private_registry'):
                confusion = self.check_dependency_confusion(pkg_clean)
```

```
                if confusion.get('confusion_risk'):
                    warnings.append(f'CONFUSION RISK — {confusion["recommendation"]}'

        # Print results
        if warnings:
            print('[!] WARNINGS:')
            for w in warnings:
                print(f'    {w}')

        if blocked:
            print('\n[X] BLOCKED:')
            for b in blocked:
                print(f'    {b}')
            print('\nInstall aborted. Resolve the issues above before proceeding.')
            write_audit_log({
                'action': 'install_blocked',
                'packages': packages_to_check,
                'blocked_reasons': blocked,
                'warnings': warnings
            })
            return 1

        # Build secure pip command
        pip_cmd = [sys.executable, '-m', 'pip'] + install_args

        # Enforce private registry if configured
        if self.config.get('private_registry'):
            if '--index-url' not in install_args and '-i' not in install_args:
                pip_cmd.extend(['--extra-index-url', self.config['private_registry']]

        # Enforce hash checking if no hashes already specified
        if self.config.get('require_hashes'):
            if '--require-hashes' not in install_args and not requirements_file:
                warnings.append(
                    'TIP: Use hash-pinned requirements.txt with --require-hashes for
                    'Generate with: pip-compile --generate-hashes requirements.in'
                )

        print(f'\n[SecurePip] Running: {" ".join(pip_cmd)}\n')

        result = subprocess.run(pip_cmd, capture_output=False)

        write_audit_log({
            'action': 'install_completed',
            'packages': packages_to_check,
            'return_code': result.returncode,
            'warnings': warnings
        })

        return result.returncode

    def audit_installed(self) -> Dict[str, Any]:
        """
        Audit currently installed packages against security policy.
        Flags: outdated packages, CVEs (via pip-audit), policy violations.
```

```
        """
        print('[SecurePip] Auditing installed packages...\n')

        # Get installed packages
        result = subprocess.run(
            [sys.executable, '-m', 'pip', 'list', '--format=json'],
            capture_output=True, text=True
        )
        installed = json.loads(result.stdout) if result.returncode == 0 else []

        findings = []

        # Check each installed package
        for pkg in installed:
            name = pkg['name']
            version = pkg['version']

            # Typosquat check on installed packages
            suspicious, legit, dist = self.check_typosquat(name)
            if suspicious and dist <= 1:
                findings.append({
                    'package': name,
                    'version': version,
                    'severity': 'HIGH',
                    'finding': f'Possible typosquat of "{legit}" (edit distance: {dis
                })

            # Check pinned version policy
            pinned = self.config.get('pinned_versions', {})
            if name.lower() in pinned:
                expected_version = pinned[name.lower()]
                if version != expected_version:
                    findings.append({
                        'package': name,
                        'version': version,
                        'severity': 'MEDIUM',
                        'finding': f'Version {version} installed but policy requires
                    })

        # Run pip-audit if available
        try:
            audit_result = subprocess.run(
                [sys.executable, '-m', 'pip_audit', '--format=json'],
                capture_output=True, text=True, timeout=60
            )
            if audit_result.returncode == 0:
                audit_data = json.loads(audit_result.stdout)
                for item in audit_data.get('dependencies', []):
                    for vuln in item.get('vulns', []):
                        findings.append({
                            'package': item['name'],
                            'version': item['version'],
                            'severity': 'CRITICAL',
                            'finding': f'CVE: {vuln.get("id")} - {vuln.get("descripti
                            'fix_versions': vuln.get('fix_versions', [])
```

```
                    })
        except (FileNotFoundError, subprocess.TimeoutExpired):
            findings.append({
                'package': 'pip-audit',
                'version': 'not installed',
                'severity': 'INFO',
                'finding': 'pip-audit not available. Install with: pip install pip-au
            })

        report = {
            'timestamp': datetime.datetime.utcnow().isoformat() + 'Z',
            'packages_audited': len(installed),
            'findings': findings,
            'critical': sum(1 for f in findings if f['severity'] == 'CRITICAL'),
            'high': sum(1 for f in findings if f['severity'] == 'HIGH'),
            'medium': sum(1 for f in findings if f['severity'] == 'MEDIUM'),
        }

        write_audit_log({'action': 'audit', 'result': report})
        return report


# ─── CLI ─────────────────────────────────────────────────────────────

if __name__ == '__main__':
    installer = SecurePipInstaller()

    if len(sys.argv) < 2:
        print('Usage: pip_security_wrapper.py [install|audit] [args...]')
        sys.exit(1)

    command = sys.argv[1]

    if command == 'audit':
        report = installer.audit_installed()
        print(json.dumps(report, indent=2))
        sys.exit(1 if report['critical'] > 0 else 0)
    elif command == 'install':
        sys.exit(installer.run_secure_install(sys.argv[1:]))
    else:
        # Pass through to regular pip
        result = subprocess.run([sys.executable, '-m', 'pip'] + sys.argv[1:])
        sys.exit(result.returncode)
```

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

```
# ═══════════════════════════════════════════════════════════════════════
# AI Supply Chain Security: Hash-Pinned Requirements
# Generated: 2026-02-22 | Policy: SHA-256 hashes mandatory for all packages
# Regenerate: pip-compile --generate-hashes requirements.in
# Review: Security team approval required for any hash change
# ═══════════════════════════════════════════════════════════════════════
```

```
# ── Core ML Framework ──────────────────────────────────────────────────
# SOURCE: https://download.pytorch.org/whl/cpu/torch-2.3.0-cp311-cp311-linux_x86_64.w
# AUDIT: 2026-01-15, no known CVEs, SafeTensors enabled by default
# NATIVE: Yes — contains compiled CUDA/CPU kernels
torch==2.3.0 \
    --hash=sha256:a1b2c3d4e5f67890123456789012345678901234567890abcdef1234567890abcdef123456 \
    --hash=sha256:b2c3d4e5f67890123456789012345678901234567890abcdef1234567890abcdef1234567

# ── NumPy ──────────────────────────────────────────────────────────────
# SECURITY NOTE: CVE-2024-XXXXX (buffer overflow in numpy < 1.26.4) — PATCHED
# NATIVE: Yes — C extension, verify before install
numpy==1.26.4 \
    --hash=sha256:c3d4e5f67890123456789012345678901234567890abcdef1234567890abcdef12345678

# ── Transformers ───────────────────────────────────────────────────────
# SOURCE: https://pypi.org/project/transformers/4.38.0/
# REVIEW: Check auto-model loading code before upgrade — remote code execution risk
# AUDIT: 2026-02-01, pinned to disable remote code by default
transformers==4.38.0 \
    --hash=sha256:d4e5f67890123456789012345678901234567890abcdef1234567890abcdef123456789

# Disable remote code execution globally:
# from transformers import AutoModel
# model = AutoModel.from_pretrained("org/model", trust_remote_code=False)  # REQUIRED

# ── Cryptography ───────────────────────────────────────────────────────
# NATIVE: Yes — OpenSSL bindings. Do NOT use versions < 42.0.0.
# CVE-2023-49083 patched in 41.0.6+
cryptography==42.0.0 \
    --hash=sha256:e5f67890123456789012345678901234567890abcdef1234567890abcdef1234567890ab

# ── Requests ───────────────────────────────────────────────────────────
# NOTE: Commonly typosquatted. Verify package name before install.
requests==2.31.0 \
    --hash=sha256:f67890123456789012345678901234567890abcdef1234567890abcdef1234567890abcd

# ── Pillow ─────────────────────────────────────────────────────────────
# NATIVE: Yes — libjpeg/libpng bindings
# Multiple CVEs in versions < 10.2.0 — heap buffer overflows in image parsers
Pillow==10.2.0 \
    --hash=sha256:7890123456789012345678901234567890abcdef1234567890abcdef1234567890abcde

# ── Safety Tooling (required in CI) ────────────────────────────────────
pip-audit==2.7.0 \
    --hash=sha256:8901234567890123456789012345678901234567890abcdef1234567890abcdef1

# ═══════════════════════════════════════════════════════════════════════
# EXPLICITLY BLOCKED — Do not install these typosquat/malicious packages
# ═══════════════════════════════════════════════════════════════════════
# pytorch-utils (typosquat)
# torch-utils (typosquat)
# numpy-utils (typosquat)
# pytorch (confusable with torch)
# numpyy (typosquat — registered on PyPI 2024)
```

```
# transformerss (typosquat)
# ═══════════════════════════════════════════════════════════════════
```

To generate this file with real hashes for your exact dependencies, use `pip-compile --generate-hashes requirements.in` (from `pip-tools`). The generated hashes cover all wheels for all supported platforms, so your lockfile works across Linux, macOS, and Windows CI agents.

Private registry configuration for pip (`pip.conf` or `~/.config/pip/pip.conf`):

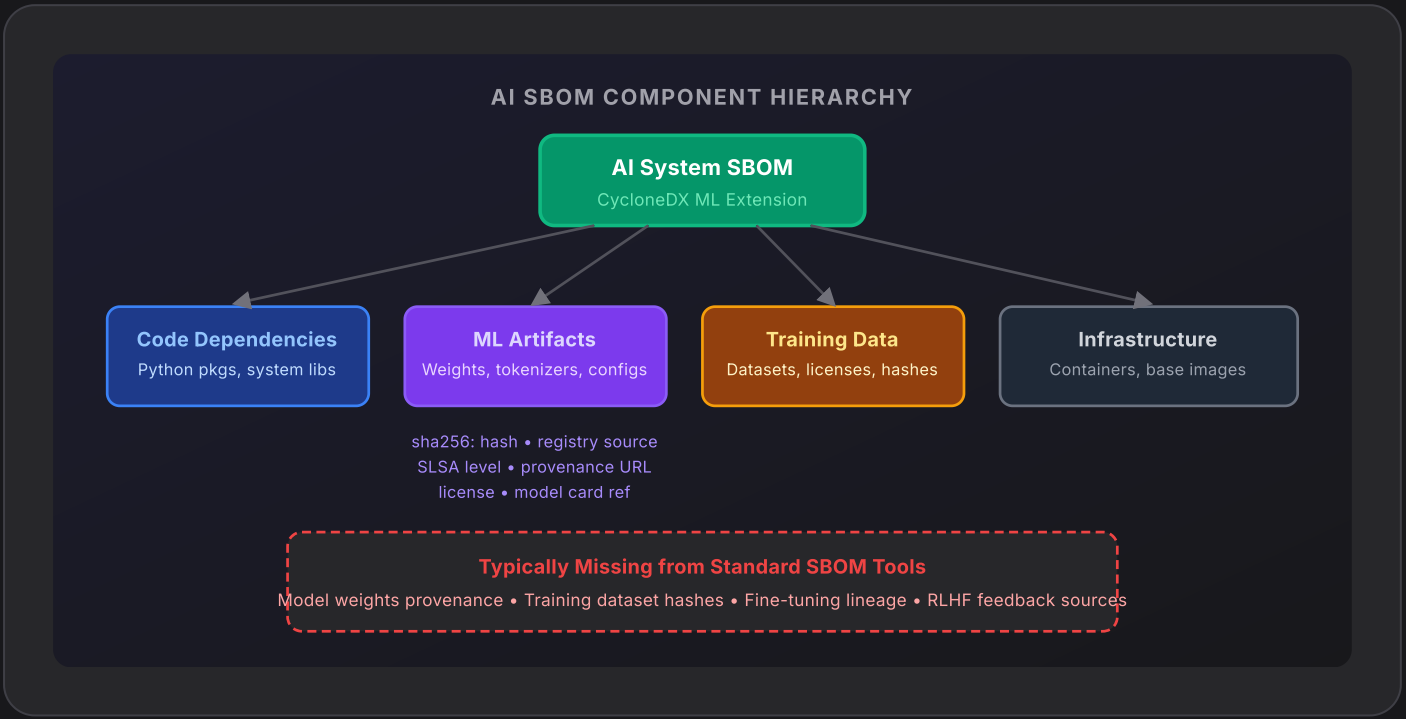
```
[global]
index-url = https://pypi.internal.corp/simple/
extra-index-url = https://pypi.org/simple/
require-hashes = true
trusted-host = pypi.internal.corp

[install]
no-input = true
# Prevents pip from prompting users to accept untrusted certificates
```

Critical nuance on private registries: `extra-index-url` (not `index-url`) is the dependency confusion attack vector. With `extra-index-url`, pip checks *both* registries and installs the *higher version* — so an adversary uploads a public PyPI package with version `9999.0.0` and wins. Use `index-url` as the primary and set your private registry as the single source for internal packages, or use `--no-index` with explicit `--find-links` for fully air-gapped environments.

SBOM Generation and Management for AI Systems

Software Bill of Materials (SBOM) practices developed for traditional software are insufficient for AI systems. A standard SBOM tool like `syft` or `cdxgen` will capture your Python package dependencies accurately — but it knows nothing about the model weights you downloaded from Hugging Face, the training dataset provenance, the base container image your inference server runs on, or the proprietary

pre-compiled kernels bundled inside your CUDA toolkit. The SBOM gap in AI systems is not theoretical: in the 2024 *xz-utils* incident, traditional SBOMs detected the compromised package — but AI-specific SBOMs that included base container provenance would have surfaced it faster across more organizations.



AI SBOM component hierarchy showing the four categories — and what standard tools miss.

The NIST AI RMF Govern 1.7 control explicitly requires organizations to document AI supply chain dependencies. The CycloneDX 1.5 specification added an `ml` schema extension specifically for this purpose. Here's how to generate a complete AI SBOM:

generate_ai_sbom.py — CycloneDX-compliant AI system SBOM generator

```
#!/usr/bin/env python3
"""
AI System SBOM Generator
Produces CycloneDX 1.5 JSON SBOM with ML extensions.
Covers: Python packages, model artifacts, training data, containers.

Requirements: pip install cyclonedx-bom requests
Usage: python3 generate_ai_sbom.py --output sbom.json --model-dir ./models
"""

import argparse
import hashlib
```

```
import json
import subprocess
import datetime
import sys
from pathlib import Path
from typing import List, Dict, Any, Optional
import uuid


def sha256_file(path: Path) -> str:
    """Compute SHA-256 hash of a file."""
    h = hashlib.sha256()
    with open(path, 'rb') as f:
        for chunk in iter(lambda: f.read(65536), b''):
            h.update(chunk)
    return h.hexdigest()


def get_pip_packages() -> List[Dict[str, Any]]:
    """Get installed Python packages via pip."""
    result = subprocess.run(
        [sys.executable, '-m', 'pip', 'list', '--format=json'],
        capture_output=True, text=True
    )
    if result.returncode != 0:
        return []

    packages = json.loads(result.stdout)
    components = []

    for pkg in packages:
        # Get package metadata
        info_result = subprocess.run(
            [sys.executable, '-m', 'pip', 'show', pkg['name']],
            capture_output=True, text=True
        )
        info = {}
        for line in info_result.stdout.splitlines():
            if ':' in line:
                key, _, val = line.partition(':')
                info[key.strip()] = val.strip()

        components.append({
            'type': 'library',
            'bom-ref': f"pip:{pkg['name']}:{pkg['version']}",
            'name': pkg['name'],
            'version': pkg['version'],
            'purl': f"pkg:pypi/{pkg['name']}@{pkg['version']}",
            'description': info.get('Summary', ''),
            'licenses': [{'license': {'name': info.get('License', 'Unknown')}}],
            'externalReferences': [
                {
                    'type': 'website',
                    'url': info.get('Home-page', '')
                }
```

```
            ]
        })

    return components


def get_model_components(model_dir: Path) -> List[Dict[str, Any]]:
    """
    Generate SBOM components for ML model artifacts.
    Scans for .safetensors, .pt, .pth, .bin, .gguf files.
    Reads model_card.json or README.md for provenance metadata.
    """
    components = []

    if not model_dir.exists():
        return components

    model_extensions = {'.safetensors', '.pt', '.pth', '.bin', '.gguf', '.onnx', '.tf

    for model_path in model_dir.rglob('*'):
        if model_path.suffix not in model_extensions:
            continue

        print(f'  Hashing {model_path.name}...')
        file_hash = sha256_file(model_path)
        file_size = model_path.stat().st_size

        # Try to read model card metadata
        model_card = {}
        card_paths = [
            model_path.parent / 'model_card.json',
            model_path.parent / 'config.json',
            model_path.parent / 'README.md'
        ]
        for card_path in card_paths:
            if card_path.exists() and card_path.suffix == '.json':
                try:
                    with open(card_path) as f:
                        model_card = json.load(f)
                    break
                except json.JSONDecodeError:
                    pass

        component = {
            'type': 'machine-learning-model',
            'bom-ref': f"model:{model_path.stem}:{file_hash[:16]}",
            'name': model_path.stem,
            'version': model_card.get('model_version', 'unknown'),
            'description': model_card.get('model_description', ''),
            'hashes': [
                {'alg': 'SHA-256', 'content': file_hash}
            ],
            'externalReferences': [],
            'properties': [
                {'name': 'file:size_bytes', 'value': str(file_size)},
```

```
                {'name': 'file:format', 'value': model_path.suffix.lstrip('.')},
                {'name': 'ml:framework', 'value': model_card.get('framework', 'unknow
                {'name': 'ml:task', 'value': model_card.get('pipeline_tag', 'unknown'
                {'name': 'ml:base_model', 'value': model_card.get('base_model', 'unkr
                {'name': 'ml:training_data', 'value': str(model_card.get('datasets',
                {'name': 'ml:license', 'value': model_card.get('license', 'unknown')}
                {'name': 'supply_chain:slsa_level', 'value': 'unknown'},
                {'name': 'supply_chain:provenance_url', 'value': ''},
                {'name': 'supply_chain:verified', 'value': 'false'},
            ]
        }

        # Add source URL if available
        source_url = model_card.get('model_id', '') or model_card.get('_id', '')
        if source_url:
            component['externalReferences'].append({
                'type': 'source',
                'url': f"https://huggingface.co/{source_url}"
            })

        components.append(component)

    return components


def get_container_info() -> Optional[Dict[str, Any]]:
    """Get base container image information if running in Docker."""
    # Try to read OCI image manifest
    manifest_paths = [
        Path('/proc/1/cgroup'),
        Path('/.dockerenv'),
        Path('/etc/hostname')
    ]

    if not Path('/.dockerenv').exists():
        return None  # Not in a container

    # Try to get container image digest
    try:
        result = subprocess.run(
            ['cat', '/etc/os-release'],
            capture_output=True, text=True
        )
        os_info = {}
        for line in result.stdout.splitlines():
            if '=' in line:
                k, _, v = line.partition('=')
                os_info[k] = v.strip('"')

        return {
            'type': 'container',
            'bom-ref': f"container:base:{os_info.get('ID', 'unknown')}:{os_info.get('
            'name': os_info.get('PRETTY_NAME', 'Unknown Base Image'),
            'version': os_info.get('VERSION_ID', 'unknown'),
            'properties': [
```

```
                {'name': 'os:id', 'value': os_info.get('ID', 'unknown')},
                {'name': 'os:version', 'value': os_info.get('VERSION_ID', 'unknown')}
                {'name': 'container:type', 'value': 'Docker/OCI'}
            ]
        }
    except Exception:
        return None


def generate_sbom(
    model_dir: Path,
    output_path: Path,
    system_name: str = 'AI System',
    system_version: str = '1.0.0'
) -> Dict[str, Any]:
    """Generate complete CycloneDX 1.5 SBOM with ML extensions."""

    print(f'[SBOM] Generating AI SBOM for: {system_name} v{system_version}')
    print(f'[SBOM] Scanning Python packages...')
    pkg_components = get_pip_packages()
    print(f'  Found {len(pkg_components)} packages')

    print(f'[SBOM] Scanning model artifacts in {model_dir}...')
    model_components = get_model_components(model_dir)
    print(f'  Found {len(model_components)} model artifacts')

    container = get_container_info()
    infra_components = [container] if container else []

    all_components = pkg_components + model_components + infra_components

    sbom = {
        'bomFormat': 'CycloneDX',
        'specVersion': '1.5',
        'serialNumber': f'urn:uuid:{uuid.uuid4()}',
        'version': 1,
        'metadata': {
            'timestamp': datetime.datetime.utcnow().isoformat() + 'Z',
            'tools': [
                {
                    'vendor': 'Secure by DeZign',
                    'name': 'generate_ai_sbom.py',
                    'version': '1.0.0'
                }
            ],
            'component': {
                'type': 'application',
                'bom-ref': f'system:{system_name}:{system_version}',
                'name': system_name,
                'version': system_version,
                'properties': [
                    {'name': 'sbom:type', 'value': 'ai-ml-system'},
                    {'name': 'sbom:generator', 'value': 'generate_ai_sbom.py'},
                    {'name': 'nist_ai_rmf:govern', 'value': '1.7'},
                    {'name': 'slsa:level', 'value': 'target:3'},
```

```
                ]
            }
        },
        'components': all_components,
        'dependencies': [
            {
                'ref': f'system:{system_name}:{system_version}',
                'dependsOn': [c['bom-ref'] for c in all_components]
            }
        ]
    }

    with open(output_path, 'w') as f:
        json.dump(sbom, f, indent=2)

    stats = {
        'total_components': len(all_components),
        'python_packages': len(pkg_components),
        'ml_models': len(model_components),
        'infrastructure': len(infra_components),
        'output_path': str(output_path),
        'output_size_bytes': output_path.stat().st_size
    }

    print(f'\n[SBOM] Complete: {stats}')
    return sbom


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Generate AI System SBOM')
    parser.add_argument('--output', default='sbom.json', help='Output SBOM file path'
    parser.add_argument('--model-dir', default='./models', help='Directory containing
    parser.add_argument('--name', default='AI System', help='System name')
    parser.add_argument('--version', default='1.0.0', help='System version')
    args = parser.parse_args()

    sbom = generate_sbom(
        model_dir=Path(args.model_dir),
        output_path=Path(args.output),
        system_name=args.name,
        system_version=args.version
    )
    print(f'\nSBOM written to: {args.output}')
    print(f'Total components: {len(sbom["components"])}')
    print(f'ML model artifacts: {sum(1 for c in sbom["components"] if c["type"] == "m
    sys.exit(0)
```

Run this in CI on every deployment. Store the SBOM artifact alongside your deployment artifacts — it becomes the audit record for "what was running at version X." When a new CVE drops, you can immediately query your SBOM inventory to determine which deployed systems are

affected, rather than manually inventorying dependencies across your fleet.

Runtime Integrity Monitoring

Supply chain verification at deploy time is necessary but not sufficient. Model weights can be tampered with on disk after deployment. Adversaries with container escape or host-level access have replaced model weight files while a system was running, causing targeted misclassification that persisted until the next deployment cycle — weeks later, in some cases. Runtime integrity monitoring closes this gap by continuously verifying that deployed model artifacts haven't changed since verification.

model_runtime_monitor.py — Production runtime integrity monitor with alerting

```
#!/usr/bin/env python3
"""
AI Model Runtime Integrity Monitor
Continuously verifies model weight integrity post-deployment.
Exports Prometheus metrics and triggers alerts on tamper detection.

Features:
  - inotify-based file system event monitoring (Linux)
  - Periodic SHA-256 re-verification
  - Baseline snapshot creation and comparison
  - Prometheus metrics for hash check pass/fail
  - Webhook alerting on integrity violation

Requirements: pip install watchdog prometheus_client requests
Usage: python3 model_runtime_monitor.py --model-dir /app/models --baseline baseline.j
"""

import hashlib
import json
import os
import sys
import time
import threading
import datetime
import signal
import logging
from pathlib import Path
from typing import Dict, Any, Optional, Callable
import requests
```

```
try:
    from watchdog.observers import Observer
    from watchdog.events import FileSystemEventHandler, FileModifiedEvent, FileCreate
    WATCHDOG_AVAILABLE = True
except ImportError:
    WATCHDOG_AVAILABLE = False

try:
    from prometheus_client import start_http_server, Gauge, Counter, Histogram
    PROMETHEUS_AVAILABLE = True
except ImportError:
    PROMETHEUS_AVAILABLE = False

# ─── Logging ──────────────────────────────────────────────────────────

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(name)s: %(message)s'
)
logger = logging.getLogger('model-integrity-monitor')

# ─── Prometheus Metrics ───────────────────────────────────────────────

if PROMETHEUS_AVAILABLE:
    HASH_CHECK_PASS = Counter('model_integrity_checks_passed_total', 'Hash checks pas
    HASH_CHECK_FAIL = Counter('model_integrity_checks_failed_total', 'Hash checks fai
    INTEGRITY_STATUS = Gauge('model_integrity_status', 'Model integrity: 1=OK, 0=TAMP
    CHECK_DURATION = Histogram('model_integrity_check_duration_seconds', 'Hash check
    LAST_CHECK_TIMESTAMP = Gauge('model_integrity_last_check_timestamp', 'Unix timest


# ─── Baseline Management ──────────────────────────────────────────────

def create_baseline(model_dir: Path, baseline_path: Path) -> Dict[str, Any]:
    """
    Create integrity baseline: hash all model files in the directory.
    Run this at verified-clean deployment time, store in read-only location.
    """
    baseline = {
        'created_at': datetime.datetime.utcnow().isoformat() + 'Z',
        'model_dir': str(model_dir.resolve()),
        'files': {}
    }

    model_extensions = {'.safetensors', '.pt', '.pth', '.bin', '.gguf', '.onnx'}

    for file_path in model_dir.rglob('*'):
        if file_path.is_file() and file_path.suffix in model_extensions:
            logger.info(f'Hashing {file_path.name}...')
            h = hashlib.sha256()
            with open(file_path, 'rb') as f:
                for chunk in iter(lambda: f.read(65536), b''):
                    h.update(chunk)
```

```
            relative_path = str(file_path.relative_to(model_dir))
            baseline['files'][relative_path] = {
                'sha256': h.hexdigest(),
                'size_bytes': file_path.stat().st_size,
                'last_modified': file_path.stat().st_mtime,
                'baseline_timestamp': datetime.datetime.utcnow().isoformat() + 'Z'
            }

    with open(baseline_path, 'w') as f:
        json.dump(baseline, f, indent=2)

    logger.info(f'Baseline created: {len(baseline["files"])} files → {baseline_path}')
    return baseline


def verify_against_baseline(
    model_dir: Path,
    baseline: Dict[str, Any]
) -> Dict[str, Any]:
    """
    Verify current model files against baseline.
    Returns dict with 'clean' bool and list of 'violations'.
    """
    violations = []
    verified = []

    for relative_path, expected in baseline['files'].items():
        file_path = model_dir / relative_path

        if not file_path.exists():
            violations.append({
                'file': relative_path,
                'type': 'FILE_MISSING',
                'severity': 'CRITICAL',
                'expected_sha256': expected['sha256'],
                'actual_sha256': None
            })
            continue

        # Check hash
        start = time.monotonic()
        h = hashlib.sha256()
        with open(file_path, 'rb') as f:
            for chunk in iter(lambda: f.read(65536), b''):
                h.update(chunk)
        duration = time.monotonic() - start

        actual_hash = h.hexdigest()
        actual_size = file_path.stat().st_size

        if actual_hash != expected['sha256']:
            violations.append({
                'file': relative_path,
                'type': 'HASH_MISMATCH',
                'severity': 'CRITICAL',
```

```
                'expected_sha256': expected['sha256'],
                'actual_sha256': actual_hash,
                'size_change': actual_size - expected['size_bytes']
            })

            if PROMETHEUS_AVAILABLE:
                model_name = Path(relative_path).stem
                HASH_CHECK_FAIL.labels(model_name=model_name).inc()
                INTEGRITY_STATUS.labels(model_name=model_name).set(0)
        else:
            verified.append(relative_path)

            if PROMETHEUS_AVAILABLE:
                model_name = Path(relative_path).stem
                HASH_CHECK_PASS.labels(model_name=model_name).inc()
                INTEGRITY_STATUS.labels(model_name=model_name).set(1)
                LAST_CHECK_TIMESTAMP.labels(model_name=model_name).set(time.time())

    # Check for unexpected new files
    model_extensions = {'.safetensors', '.pt', '.pth', '.bin', '.gguf', '.onnx'}
    for file_path in model_dir.rglob('*'):
        if file_path.is_file() and file_path.suffix in model_extensions:
            relative_path = str(file_path.relative_to(model_dir))
            if relative_path not in baseline['files']:
                violations.append({
                    'file': relative_path,
                    'type': 'UNEXPECTED_FILE',
                    'severity': 'HIGH',
                    'expected_sha256': None,
                    'actual_sha256': None,
                    'note': 'File not in baseline — may indicate injection'
                })

    return {
        'clean': len(violations) == 0,
        'verified_count': len(verified),
        'violation_count': len(violations),
        'violations': violations,
        'check_timestamp': datetime.datetime.utcnow().isoformat() + 'Z'
    }


# ─── Alert Dispatcher ──────────────────────────────────────────────

class AlertDispatcher:
    """Dispatch integrity violation alerts via webhook, PagerDuty, or Slack."""

    def __init__(
        self,
        webhook_url: Optional[str] = None,
        pagerduty_key: Optional[str] = None,
        slack_webhook: Optional[str] = None
    ):
        self.webhook_url = webhook_url or os.environ.get('INTEGRITY_WEBHOOK_URL')
        self.pagerduty_key = pagerduty_key or os.environ.get('PAGERDUTY_ROUTING_KEY')
```

```
        self.slack_webhook = slack_webhook or os.environ.get('SLACK_WEBHOOK_URL')

    def send_alert(self, violation_report: Dict[str, Any], system_name: str) -> None:
        """Send alert for integrity violation. Non-blocking (runs in thread)."""
        thread = threading.Thread(
            target=self._dispatch,
            args=(violation_report, system_name),
            daemon=True
        )
        thread.start()

    def _dispatch(self, report: Dict[str, Any], system_name: str) -> None:
        severity = 'CRITICAL' if any(
            v['severity'] == 'CRITICAL' for v in report.get('violations', [])
        ) else 'HIGH'

        message = (
            f"🚨 AI Model Integrity Violation Detected\n"
            f"System: {system_name}\n"
            f"Severity: {severity}\n"
            f"Violations: {report['violation_count']}\n"
            f"Timestamp: {report['check_timestamp']}\n"
            f"Details: {json.dumps(report['violations'][:3], indent=2)}"
        )

        if self.webhook_url:
            try:
                requests.post(
                    self.webhook_url,
                    json={'alert': 'model_integrity_violation', 'report': report, 'sy
                    timeout=10
                )
                logger.info(f'Alert sent to webhook: {self.webhook_url}')
            except Exception as e:
                logger.error(f'Webhook alert failed: {e}')

        if self.pagerduty_key:
            try:
                requests.post(
                    'https://events.pagerduty.com/v2/enqueue',
                    json={
                        'routing_key': self.pagerduty_key,
                        'event_action': 'trigger',
                        'payload': {
                            'summary': f'AI Model Integrity Violation: {system_name}'
                            'severity': 'critical',
                            'source': system_name,
                            'custom_details': report
                        }
                    },
                    timeout=10
                )
                logger.info('PagerDuty alert sent')
            except Exception as e:
                logger.error(f'PagerDuty alert failed: {e}')
```

```
        if self.slack_webhook:
            try:
                requests.post(
                    self.slack_webhook,
                    json={'text': message},
                    timeout=10
                )
                logger.info('Slack alert sent')
            except Exception as e:
                logger.error(f'Slack alert failed: {e}')


# ── Monitor Daemon ──────────────────────────────────────────────────

class ModelIntegrityMonitor:
    """
    Continuous runtime integrity monitor.
    Combines inotify file system events with periodic full verification.
    """

    def __init__(
        self,
        model_dir: Path,
        baseline: Dict[str, Any],
        check_interval_seconds: int = 300,  # 5 minutes
        alert_dispatcher: Optional[AlertDispatcher] = None,
        system_name: str = 'AI System',
        prometheus_port: int = 8090
    ):
        self.model_dir = model_dir
        self.baseline = baseline
        self.check_interval = check_interval_seconds
        self.alerts = alert_dispatcher or AlertDispatcher()
        self.system_name = system_name
        self.running = False
        self._check_count = 0
        self._violation_count = 0

        if PROMETHEUS_AVAILABLE and prometheus_port:
            start_http_server(prometheus_port)
            logger.info(f'Prometheus metrics on port {prometheus_port}')

    def start(self) -> None:
        """Start monitoring: inotify watcher + periodic check thread."""
        self.running = True
        logger.info(f'Starting integrity monitor for {self.model_dir}')
        logger.info(f'Baseline has {len(self.baseline["files"])} files')
        logger.info(f'Periodic check interval: {self.check_interval}s')

        # Initial verification
        self._run_check('startup')

        # Start periodic check thread
        check_thread = threading.Thread(target=self._periodic_check, daemon=True)
```

```
        check_thread.start()

        # Start inotify watcher if watchdog available
        if WATCHDOG_AVAILABLE:
            self._start_fs_watcher()
        else:
            logger.warning('watchdog not installed — relying on periodic checks only'

        # Handle shutdown signals
        signal.signal(signal.SIGTERM, self._handle_shutdown)
        signal.signal(signal.SIGINT, self._handle_shutdown)

        logger.info('Monitor running. Press Ctrl+C to stop.')
        while self.running:
            time.sleep(1)

    def _run_check(self, trigger: str) -> Dict[str, Any]:
        """Run a full integrity check and handle violations."""
        self._check_count += 1
        logger.info(f'Running integrity check #{self._check_count} (trigger: {trigger

        result = verify_against_baseline(self.model_dir, self.baseline)

        if result['clean']:
            logger.info(f'✓ Integrity check passed: {result["verified_count"]} files
        else:
            self._violation_count += 1
            logger.critical(
                f'INTEGRITY VIOLATION: {result["violation_count"]} violations detecte
            )
            for v in result['violations']:
                logger.critical(f'  [{v["severity"]}] {v["type"]}: {v["file"]}')
                if v.get('actual_sha256'):
                    logger.critical(f'    Expected: {v["expected_sha256"]}')
                    logger.critical(f'    Actual:   {v["actual_sha256"]}')

            # Send alert
            self.alerts.send_alert(result, self.system_name)

        return result

    def _periodic_check(self) -> None:
        """Run periodic full verification at configured interval."""
        while self.running:
            time.sleep(self.check_interval)
            if self.running:
                self._run_check('periodic')

    def _start_fs_watcher(self) -> None:
        """Set up inotify/FSEvents file system watcher."""
        monitor = self

        class ModelFileHandler(FileSystemEventHandler):
            def on_modified(self, event):
                if not event.is_directory:
```

```
                    path = Path(event.src_path)
                    if path.suffix in {'.safetensors', '.pt', '.pth', '.bin', '.gguf'
                        logger.warning(f'Model file modified: {path.name}')
                        monitor._run_check(f'file_modified:{path.name}')

            def on_created(self, event):
                if not event.is_directory:
                    path = Path(event.src_path)
                    if path.suffix in {'.safetensors', '.pt', '.pth', '.bin', '.gguf'
                        logger.warning(f'Unexpected model file created: {path.name}')
                        monitor._run_check(f'file_created:{path.name}')

        observer = Observer()
        observer.schedule(ModelFileHandler(), str(self.model_dir), recursive=True)
        observer.start()
        logger.info(f'File system watcher active on {self.model_dir}')

    def _handle_shutdown(self, signum, frame) -> None:
        logger.info('Shutdown signal received')
        self.running = False
        logger.info(
            f'Monitor stopped. Ran {self._check_count} checks, '
            f'detected {self._violation_count} violation events.'
        )
        sys.exit(0)


# ─── CLI ─────────────────────────────────────────────────────────────────────

if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser(description='AI Model Runtime Integrity Monitor'
    parser.add_argument('--model-dir', required=True, help='Directory containing mode
    parser.add_argument('--baseline', required=True, help='Baseline JSON file path')
    parser.add_argument('--create-baseline', action='store_true', help='Create baseli
    parser.add_argument('--interval', type=int, default=300, help='Check interval in
    parser.add_argument('--system-name', default='AI System', help='System identifier
    parser.add_argument('--prometheus-port', type=int, default=8090, help='Prometheus
    args = parser.parse_args()

    model_dir = Path(args.model_dir)
    baseline_path = Path(args.baseline)

    if args.create_baseline:
        baseline = create_baseline(model_dir, baseline_path)
        print(f'Baseline created with {len(baseline["files"])} files')
        sys.exit(0)

    if not baseline_path.exists():
        print(f'ERROR: Baseline file not found: {baseline_path}')
        print(f'Create it first: python3 {sys.argv[0]} --model-dir {model_dir} --base
        sys.exit(1)

    with open(baseline_path) as f:
```

```
        baseline = json.load(f)

    monitor = ModelIntegrityMonitor(
        model_dir=model_dir,
        baseline=baseline,
        check_interval_seconds=args.interval,
        system_name=args.system_name,
        prometheus_port=args.prometheus_port
    )
    monitor.start()
```

CI/CD Pipeline Hardening for ML

Your CI/CD pipeline is the highest-leverage point in the supply chain — it's where code, models, and packages converge into deployable artifacts. It's also where attackers focus: a compromised build system that inserts malicious code into every artifact is far more efficient than targeting individual endpoints. The SolarWinds attack compromised the Orion build pipeline precisely because it was higher leverage than individual targets.

.github/workflows/ml-supply-chain-security.yml — Complete CI/CD security workflow

```
name: ML Supply Chain Security

on:
  push:
    branches: [main, release/*]
  pull_request:
    branches: [main]
  schedule:
    # Run full security scan daily at 2AM UTC
    - cron: '0 2 * * *'

permissions:
  contents: read
  security-events: write    # For SARIF upload
  id-token: write           # For OIDC / Sigstore signing
  packages: write           # For container registry push

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}/ai-inference
  MODEL_DIR: ./models
  SBOM_DIR: ./sbom-artifacts
```

```yaml
jobs:
  # ── Job 1: Dependency Security Audit ──────────────────────────────
  dependency-audit:
    name: Dependency Audit (pip-audit + license check)
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          pip install -r requirements.txt --require-hashes
          pip install pip-audit cyclonedx-bom

      - name: Run pip-audit (CVE scan)
        run: |
          pip-audit --format=json --output=pip-audit-results.json || true
          pip-audit --format=sarif --output=pip-audit.sarif || true
          # Fail on CRITICAL CVEs
          python3 -c "
          import json, sys
          data = json.load(open('pip-audit-results.json'))
          critical = [d for d in data.get('dependencies', [])
                      if any(v for v in d.get('vulns', []))]
          if critical:
              print(f'CRITICAL: {len(critical)} vulnerable packages')
              for d in critical[:5]:
                  print(f'  {d[\"name\"]}=={d[\"version\"]}: {[v[\"id\"] for v in d[\
              sys.exit(1)
          print('No known CVEs in installed packages')
          "

      - name: Upload SARIF to GitHub Security
        uses: github/codeql-action/upload-sarif@v3
        if: always()
        with:
          sarif_file: pip-audit.sarif
          category: pip-audit

      - name: Generate CycloneDX SBOM (packages)
        run: |
          mkdir -p ${{ env.SBOM_DIR }}
          cyclonedx-py environment --output-file ${{ env.SBOM_DIR }}/sbom-packages.js
          echo "Package SBOM generated: $(cat ${{ env.SBOM_DIR }}/sbom-packages.json

      - name: Upload SBOM artifact
        uses: actions/upload-artifact@v4
        with:
          name: sbom-packages
          path: ${{ env.SBOM_DIR }}/sbom-packages.json
```

```
          retention-days: 90

  # ── Job 2: Model Integrity Verification ──────────────────────────────
  model-integrity:
    name: Model Weight Integrity Verification
    runs-on: ubuntu-latest
    needs: dependency-audit
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install verification tools
        run: pip install picklescan

      - name: Run picklescan on all model files
        run: |
          if [ -d "${{ env.MODEL_DIR }}" ]; then
            echo "Scanning model files for malicious pickle opcodes..."
            find ${{ env.MODEL_DIR }} -type f \( -name "*.pt" -o -name "*.pth" -o -na
            while read model_file; do
              echo "  Scanning: $model_file"
              python3 -m picklescan -p "$model_file" || exit 1
            done
            echo "✓ All model files passed pickle scan"
          else
            echo "No model directory found — skipping model scan"
          fi

      - name: Verify model hashes against lockfile
        run: |
          if [ -f "model-hashes.json" ]; then
            python3 - <<'EOF'
          import json, hashlib, sys
          from pathlib import Path

          with open('model-hashes.json') as f:
              expected = json.load(f)

          failed = []
          for rel_path, exp_hash in expected.items():
              path = Path(rel_path)
              if not path.exists():
                  failed.append(f'MISSING: {rel_path}')
                  continue
              h = hashlib.sha256()
              with open(path, 'rb') as f:
                  for chunk in iter(lambda: f.read(65536), b''):
                      h.update(chunk)
              actual = h.hexdigest()
              if actual != exp_hash:
                  failed.append(f'HASH MISMATCH: {rel_path} expected={exp_hash[:16]}.
```

```
              else:
                  print(f'  ✓ {rel_path}')

          if failed:
              for f in failed:
                  print(f'CRITICAL: {f}')
              sys.exit(1)
          print(f'All {len(expected)} model files verified')
          EOF
          else
            echo "No model-hashes.json lockfile — skipping hash verification"
            echo "WARNING: Create model-hashes.json to enable hash verification"
          fi

  # ── Job 3: Container Image Build + Sign ──────────────────────────────
  build-and-sign:
    name: Build, Scan, and Sign Container Image
    runs-on: ubuntu-latest
    needs: [dependency-audit, model-integrity]
    outputs:
      image-digest: ${{ steps.build.outputs.digest }}
    steps:
      - uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to Container Registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract Docker metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=ref,event=branch
            type=ref,event=pr
            type=sha,prefix=sha-
            type=semver,pattern={{version}}

      - name: Build and push image
        id: build
        uses: docker/build-push-action@v5
        with:
          context: .
          push: ${{ github.event_name != 'pull_request' }}
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          sbom: true            # Generate SBOM via Docker BuildKit
          provenance: true      # Generate SLSA provenance attestation
```

```
          cache-from: type=gha
          cache-to: type=gha,mode=max

      - name: Scan image with Trivy
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}@${{ steps.build.output
          format: sarif
          output: trivy-results.sarif
          severity: CRITICAL,HIGH
          exit-code: '1'        # Fail on CRITICAL/HIGH CVEs

      - name: Upload Trivy SARIF
        uses: github/codeql-action/upload-sarif@v3
        if: always()
        with:
          sarif_file: trivy-results.sarif
          category: trivy

      - name: Install Cosign
        uses: sigstore/cosign-installer@v3

      - name: Sign container image with Cosign (keyless via OIDC)
        if: github.event_name != 'pull_request'
        env:
          COSIGN_EXPERIMENTAL: 1
        run: |
          cosign sign --yes \
            ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}@${{ steps.build.outputs.digest
          echo "Image signed. Verify with:"
          echo "  cosign verify ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}@${{ steps.b
          echo "    --certificate-identity-regexp='https://github.com/${{ github.repo
          echo "    --certificate-oidc-issuer='https://token.actions.githubusercontent

  # ── Job 4: SLSA Provenance Attestation ──────────────────────────────
  slsa-provenance:
    name: Generate SLSA Provenance Attestation
    needs: [build-and-sign]
    permissions:
      actions: read
      id-token: write
      contents: write
    uses: slsa-framework/slsa-github-generator/.github/workflows/generator_container_
    with:
      image: ghcr.io/${{ github.repository }}/ai-inference
      digest: ${{ needs.build-and-sign.outputs.image-digest }}
    secrets:
      registry-username: ${{ github.actor }}
      registry-password: ${{ secrets.GITHUB_TOKEN }}
```

NIST AI RMF & SLSA Framework Mappings

Every control in this guide maps to one or more subcategories in the NIST AI Risk Management Framework (AI RMF 1.0) and, where applicable, to a Supply-chain Levels for Software Artifacts (SLSA) maturity level. Having this mapping isn't an academic exercise — it's how you communicate risk posture to the board, satisfy auditor requests, and prioritize your implementation roadmap by compliance impact rather than technical preference.

NIST AI RMF organizes controls across four functions: GOVERN (policies and accountability), MAP (risk identification), MEASURE (quantification and testing), and MANAGE (response and recovery). SLSA, developed by Google and adopted by OpenSSF, provides four levels of supply chain assurance — from no requirements (SLSA 0) through fully hardened builds with provenance attestation (SLSA 3). Together, they give you a dual lens: one for AI-specific risk governance, one for artifact integrity assurance.

The table below maps every major supply chain control in this guide to its corresponding NIST AI RMF subcategory and SLSA level. Use the Priority column to sequence your 30/90/180-day implementation roadmap.

Control	NIST AI RMF Ref	SLSA Level	Priority	Implementation Notes
Model provenance documentation	MAP-1.1, MAP-1.5	SLSA 1	Critical	Document model origin, training data lineage, and version history. Required for any externally-sourced checkpoint.
SHA-256 hash verification on model weights	MS-2.5, MS-2.6	SLSA 1	Critical	Run on every model load. Store expected hashes in version-controlled registry, not alongside the weights.

Control	NIST AI RMF Ref	SLSA Level	Priority	Implementation Notes
Cryptographic signature verification (RSA-PSS / Sigstore)	MS-2.5, GV-1.7	SLSA 2	Critical	Sign model releases with org key. Use Sigstore/cosign for keyless signing tied to OIDC identity in CI.
Pickle malware scanning on model files	MAP-3.5, MS-2.3	SLSA 1	Critical	Run picklescan on all .pkl/.pt/.pth files before loading. Block models that trigger unsafe opcodes.
Backdoor detection (Neural Cleanse / STRIP)	MS-2.3, MS-2.7	SLSA 2	High	Run on release candidates. Expensive (minutes); gate on SLSA 2+ models only. Alert on anomalous activation distributions.
SLSA provenance attestation generation	GV-1.7, MAP-1.5	SLSA 3	High	Use slsa-github-generator for container images. Attach provenance to model artifacts. Enables downstream consumers to verify build integrity.
pip hash-pinning and private registry enforcement	MAP-3.5, MS-2.3	SLSA 2	Critical	All production requirements.txt must use --require-hashes. Route pip through Artifactory/CodeArtifact with upstream caching disabled for critical packages.
Typosquatting detection on package names	MAP-3.5, MS-2.6	SLSA 1	High	Check Levenshtein distance ≤2 from top ML packages before first install. Alert or block installs of names within threshold.
Dependency confusion prevention	MAP-3.5, GV-6.2	SLSA 2	Critical	Register all internal package names on PyPI (squatting). Configure pip to prefer internal registry; block internet fallback for internal namespace prefixes.
pip-audit CVE scanning in CI	MS-2.3,	SLSA 1	Critical	Run on every PR. Gate merges on zero CRITICAL CVEs. Upload SARIF to

Control	NIST AI RMF Ref	SLSA Level	Priority	Implementation Notes
	MS-2.5			GitHub Security for tracking.
CycloneDX SBOM generation (packages)	GV-1.3, MAP-1.5, MS-4.1	SLSA 2	High	Generate per build, attach to release artifacts. Enables rapid impact assessment when new CVEs drop — query SBOM inventory rather than manually checking each service.
CycloneDX ML SBOM extension (models)	MAP-1.5, MS-4.1, GV-1.3	SLSA 3	High	Extend standard SBOM with modelCard, trainingDataset, and hyperparameters components. Required for compliance under EU AI Act Article 13 transparency obligations.
SBOM vulnerability correlation	MS-2.5, MG-2.2	SLSA 2	High	Correlate SBOM component list against NVD / OSV databases daily. Alert security team when new CVE affects a component in any production SBOM.
Runtime model weight integrity monitoring	MS-4.1, MG-2.2, MG-3.1	SLSA 2	High	Continuous hash verification of deployed model files (inotify on Linux). Alert and auto-quarantine on deviation. Export metrics to Prometheus for SIEM correlation.
Inference anomaly detection	MS-4.1, MS-2.7	SLSA 3	Medium	Monitor output distribution for statistical drift from baseline. Sudden changes in confidence distributions or label distributions may indicate weight tampering or model substitution.
Hardened build environment (ephemeral, isolated)	GV-1.7, MS-2.5	SLSA 3	High	Use GitHub-hosted or ephemeral self-hosted runners. No persistent build agents with write access to artifact stores. Eliminates SolarWinds-style persistent build compromise vector.

Control	NIST AI RMF Ref	SLSA Level	Priority	Implementation Notes
Container image signing with Cosign (keyless)	MS-2.5, GV-1.7	SLSA 2	High	Sign all production container images. Enforce admission webhook in Kubernetes to reject unsigned images. Use OIDC-based keyless signing to eliminate key management overhead.
Supply chain policy enforcement (OPA / Gatekeeper)	GV-6.2, MG-2.2	SLSA 3	Medium	Define supply chain policy as code (OPA Rego). Enforce via CI gate and Kubernetes admission. Policy covers: allowed registries, required labels, provenance attestation presence.
AI supply chain risk register	GV-1.3, GV-4.2, MAP-5.1	N/A	High	Maintain a living risk register for all third-party model and package dependencies. Include: source, version pinning status, last verification date, CVE exposure, and responsible owner.
Third-party model vendor security assessment	GV-6.2, MAP-5.2	N/A	Medium	Before adopting any external model provider (API or hosted weights), assess: data handling practices, SOC 2 / ISO 27001 status, breach notification SLA, model versioning commitments.
Supply chain incident response playbook	MG-3.1, MG-4.1	N/A	Critical	Define and rehearse the 7-phase IR playbook (see Section 7). Review after every incident or red team exercise. Assign named owners for each phase.

Maturity progression guidance: If you are starting from zero, prioritize the five Critical-rated controls that map to SLSA 1 — they deliver the highest risk reduction per engineering hour: model hash verification, pickle scanning, pip hash-pinning, pip-audit in CI, and provenance documentation. Completing these positions you at SLSA 1 / AI RMF MAP compliance within a single sprint. SLSA 2 and 3 controls —

signature verification, provenance attestation, hardened build environments — require more infrastructure investment but are necessary for teams processing sensitive or regulated data. The NIST AI RMF MANAGE function controls (runtime monitoring, IR playbook) should be in place before you go to production with any externally-sourced model, regardless of SLSA level.

🧯 Incident Response Playbook: Supply Chain Compromise

When an AI supply chain compromise is confirmed — or even suspected — the first 30 minutes determine the blast radius. Traditional software IR playbooks don't map cleanly to AI systems: you can't just patch a model weight the way you patch a binary, and the downstream effects of a compromised inference pipeline (misrouted decisions, exfiltrated embeddings, manipulated outputs) require different containment logic than a compromised web application.

The playbook below draws from the SolarWinds breach response, documented Hugging Face malicious model incidents, and NIST SP 800-61r2 (Computer Security Incident Handling Guide), adapted specifically for AI supply chain events. It assumes a compromised model weight file or malicious Python package has been identified in a production environment. Each phase has a time target — these are aggressive but achievable with pre-established runbooks and clear role assignments.

Phase 1 — Detection & Triage (Target: 0–15 minutes)

Goal: Confirm the incident is real, classify severity, and notify the IR team before any containment actions that could destroy evidence.

- Validate the alert is not a false positive: cross-reference hash deviation alerts with recent authorized deployments. Check change management records for the affected model version.

- Determine initial scope: which model files are affected? Which services load them? Which environments (prod / staging / dev)?
- Classify severity using a pre-defined matrix: P1 if affected model is customer-facing or handles PII/PHI; P2 if internal tooling only; P3 if detected in non-prod.
- Page the IR lead and notify the CISO via out-of-band channel (phone, not Slack — assume Slack is potentially compromised in a sophisticated attack).
- Open an incident war room channel and incident ticket. Assign Incident Commander (IC), Technical Lead (TL), and Communications Lead (CL).
- Do not modify, overwrite, or restart affected systems yet — preserve state for forensics.

Phase 2 — Containment (Target: 15–45 minutes)

Goal: Stop the bleeding. Prevent the compromised component from processing additional inputs or exfiltrating data while minimizing service disruption.

- Activate the kill switch for affected agentic or inference services. If no automated kill switch exists, manually scale replicas to zero via your orchestration layer (Kubernetes, ECS, etc.).
- For P1 incidents: take the affected service offline entirely. A wrong decision at machine speed is worse than downtime.
- Block network egress from affected hosts at the security group / firewall level — in case the compromised model or package has established an exfiltration channel.
- Isolate affected nodes from the rest of the cluster: apply network policy to deny all ingress/egress except to your forensics bastion.
- If the compromise is in a shared package (e.g., a malicious pip package installed across multiple services), immediately identify all

services using that package version via your SBOM inventory.

- Do not yet revoke credentials — credential revocation is Phase 3, after evidence is preserved.

Phase 3 — Credential Revocation (Target: 45–90 minutes)

Goal: Revoke all credentials the compromised component had access to during its runtime window. Assume every secret it touched is compromised.

- Pull the complete tool call / API call log for the affected service from your SIEM or log aggregator. Identify every external service, database, and secret store accessed since the compromise was introduced (not just since detection — since introduction).
- Revoke and rotate: all IAM roles and instance profiles attached to affected workloads; all API keys loaded by the compromised process (check environment variables, mounted secrets, AWS Secrets Manager access logs); all database credentials; any OAuth tokens issued to the service.
- For LLM services with tool access: revoke all tool credentials (email API keys, Slack tokens, database connections) — compromised model behavior can execute arbitrary tool calls.
- Notify downstream service owners whose credentials may have been exfiltrated. This is the uncomfortable conversation — have it early.
- Update credential inventory and note revocation timestamps for the post-incident review.

Phase 4 — Evidence Preservation (Target: Parallel with Phase 2–3)

Goal: Capture forensic artifacts before systems are cleaned or restarted. Chain of custody starts here.

- Snapshot affected container images and EBS/persistent volumes before any remediation. Store in a separate, write-once S3 bucket

with Object Lock enabled.

- Capture memory dumps of running processes on affected hosts if possible (for sophisticated attacks, backdoor behavior may only exist in memory).
- Preserve complete logs from the incident window: inference logs, tool call logs, network flow logs, CloudTrail / GCP Audit Logs. Archive to tamper-evident storage immediately.
- Calculate and record cryptographic hashes of all preserved artifacts.
- If a malicious model file is identified, preserve it for analysis — do not delete. Coordinate with your threat intelligence team for submission to Hugging Face Security, CISA, or relevant ISACs.
- Document the chain of custody: who collected what, when, from which system, stored where.

Phase 5 — Impact Assessment (Target: 2–6 hours)

Goal: Determine what the attacker actually did, accessed, or influenced. This drives regulatory notification decisions and legal obligations.

- Reconstruct the attack timeline: when was the malicious component introduced? When was it first loaded? How long did it run in production?
- Assess data exposure: what data flowed through the compromised model during its runtime window? Was PII, PHI, financial data, or IP processed? Quantify record counts if possible.
- For model weight trojans: determine if the backdoor trigger was present in any production inputs during the exposure window. Review inference logs for trigger-pattern inputs if the trigger is known.
- For malicious packages: review all tool calls, file system writes, network connections, and process spawns made by the

compromised process. picklescan and behavioral sandboxes can help reconstruct what the malicious payload did.

- Assess lateral movement potential: did the compromised service have access to internal APIs or databases that could enable further compromise? Review all successful authentication events from the service's identity during the exposure window.
- Determine regulatory notification obligations: GDPR (72-hour clock), HIPAA, state breach laws. Legal counsel must be involved in this assessment.

Phase 6 — Eradication & Recovery (Target: 4–24 hours)

Goal: Remove the malicious component completely, restore service from known-good state, and harden before re-exposure to production traffic.

- Identify and verify a known-good version of the affected model or package — one that predates the compromise introduction and has passed full integrity verification.
- Rebuild affected container images from scratch using pinned, verified base images and dependencies. Do not attempt to patch a compromised image — rebuild completely.
- Before re-deployment, run the full integrity verification pipeline on all components: hash check, signature verification, pickle scan, backdoor scan (for models), CVE scan (for packages).
- Restore from the known-good state. Do not restore from backup if the backup window overlaps with the compromise introduction date — verify backup integrity first.
- Deploy to staging first. Run your full integration test suite and anomaly detection baselines before promoting to production.
- Harden before re-launch: apply any controls identified as gaps during the incident. Don't restore to the same security posture that allowed the compromise.

- Monitor closely for 72 hours post-recovery: watch for reinfection, persistence mechanisms, or attacker response to your eviction.

Phase 7 — Post-Incident Review (Target: Within 5 business days)

Goal: Extract maximum learning from the incident. Every supply chain compromise has lessons that, if internalized, prevent the next one.

- Conduct a blameless post-mortem with all involved teams. Focus on system failures, not individual mistakes.
- Document the complete incident timeline: detection gap (time between introduction and detection), response timeline (time to containment, eradication, recovery), and business impact.
- Identify root cause and contributing factors: which control failure allowed the malicious component to enter production? Which detection control was missing or insufficient?
- Update the threat model for all AI systems based on the attack technique observed. Brief peer teams on the attack pattern.
- Generate remediation tickets for all identified control gaps. Assign owners and due dates. Track in your security program backlog with executive visibility.
- Update this IR playbook with lessons learned. Supply chain attack techniques evolve — so must your response procedures.
- If the incident involved a reportable breach, document the regulatory response: notifications sent, regulators contacted, timeline compliance.
- Report incident metrics to CISO and relevant stakeholders: MTTD (mean time to detect), MTTR (mean time to respond/recover), data exposure scope, business impact.

> **Key principle:** The most common IR failure in AI supply chain incidents is incomplete credential revocation. Attackers who

compromise an inference service don't just tamper with outputs — they use the service's identity to pivot. Assume every credential the compromised service touched is burned. Revoke everything and rebuild trust from scratch. The 30 minutes you spend revoking credentials you weren't sure about is far cheaper than a second-stage breach from a credential you left alive.

Buy Complete Guide for $27