HMAC Integrity Checker

```
#!/usr/bin/env python3
"""
DNS Subdomain Batch Integrity Checker

This script processes multiple message/HMAC file pairs in a directory, following the pattern:
message_#.txt and message_#.hmac

It automatically detects and verifies all matching pairs in the specified directory,
generating a comprehensive report of integrity issues across all files.

Usage:
    python dns_batch_integrity.py --directory <logs_directory> --output <output_dir>
"""

import hmac
import hashlib
import sys
import os
import re
import argparse
import json
import glob
from datetime import datetime
from typing import Dict, List, Any, Tuple, Set

# The valid signing key
VALID_KEY = 'ciCloud-API-20240315-4f7b9c'

class DNSSubdomainBatchChecker:
    def __init__(self, key: str = VALID_KEY):
        """
        Initialize the DNS Subdomain Integrity Checker.

        Args:
            key: The HMAC signing key
        """
```

```
        self.key = key

        # Initialize common DNS patterns to check for tampering
        self.common_subdomains = {
            'www', 'mail', 'api', 'admin', 'portal', 'test', 'dev', 'staging',
            'secure', 'vpn', 'internal', 'mx', 'smtp', 'pop', 'imap', 'webmail',
            'remote', 'cdn', 'dns', 'ns1', 'ns2', 'ldap', 'db', 'mysql', 'ftp'
        }

        # Suspicious TLDs often used in attacks
        self.suspicious_tlds = {
            'xyz', 'top', 'club', 'cyou', 'icu', 'rest', 'space', 'casa',
            'monster', 'bar', 'gq', 'tk', 'ml', 'cf', 'ga'
        }

        # Common character substitutions used in spoofing
        self.char_substitutions = {
            '0': 'o', 'o': '0',
            '1': 'l', 'l': '1', 'i': '1',
            '5': 's', 's': '5',
            '3': 'e', 'e': '3',
            '4': 'a', 'a': '4',
            '6': 'g', 'g': '6',
            '7': 't', 't': '7',
            '8': 'b', 'b': '8'
        }

    def calculate_hmac(self, message: str) -> str:
        """
        Calculate HMAC signature for a message.

        Args:
            message: The message to sign

        Returns:
            The HMAC signature (hex encoded)
        """
        key_bytes = self.key.encode('utf-8')
        message_bytes = message.encode('utf-8')
        signature = hmac.new(key_bytes, message_bytes, hashlib.sha256)
        return signature.hexdigest()
```

```
def verify_hmac(self, message: str, signature: str) -> bool:
    """
    Verify if a message's HMAC signature is valid.

    Args:
        message: The message to verify
        signature: The provided HMAC signature

    Returns:
        True if signature is valid, False otherwise
    """
    calculated_signature = self.calculate_hmac(message)
    # Use constant-time comparison to prevent timing attacks
    return hmac.compare_digest(calculated_signature, signature)

def read_file(self, file_path: str) -> List[str]:
    """
    Read a file and return its lines.

    Args:
        file_path: Path to the file

    Returns:
        List of lines from the file
    """
    with open(file_path, 'r') as f:
        return [line.rstrip() for line in f.readlines()]

def find_file_pairs(self, directory: str) -> List[Tuple[str, str]]:
    """
    Find matching message/HMAC file pairs in the directory.

    Args:
        directory: Directory to search for files

    Returns:
        List of tuples (message_file_path, hmac_file_path)
    """
    file_pairs = []
```

```
        # Find all message_*.txt files
        message_files = glob.glob(os.path.join(directory, "message_*.txt"))

        for message_file in message_files:
            # Extract the number part
            match = re.search(r'message_(\d+)\.txt$', message_file)
            if match:
                number = match.group(1)
                hmac_file = os.path.join(directory, f"message_{number}.hmac")

                # Check if the corresponding HMAC file exists
                if os.path.exists(hmac_file):
                    file_pairs.append((message_file, hmac_file))

        return file_pairs

    def extract_domain_info(self, log_entry: str) -> Dict[str, Any]:
        """
        Extract domain and subdomain information from a log entry.

        Args:
            log_entry: A log entry string

        Returns:
            Dictionary with extracted domain information
        """
        domain_info = {
            'has_domain': False,
            'domain': '',
            'subdomain': '',
            'tld': ''
        }

        # Try to find domain patterns in the log entry
        # This regex looks for domain.tld or subdomain.domain.tld patterns
        domain_matches = re.findall(r'([a-zA-Z0-9][-a-zA-Z0-9]*(\.[a-zA-Z0-9][-a-zA-Z0-
9]*)+)', log_entry)

        if domain_matches:
            domain_info['has_domain'] = True
            full_domain = domain_matches[0][0]
```

```
            domain_info['domain'] = full_domain

            # Split by dots to extract subdomain and TLD
            parts = full_domain.split('.')

            if len(parts) >= 2:
                domain_info['tld'] = parts[-1].lower()

                if len(parts) > 2:
                    domain_info['subdomain'] = '.'.join(parts[:-2])

        return domain_info

    def detect_tampering(self, log_entry: str) -> Dict[str, Any]:
        """
        Detect possible tampering in a DNS log entry.

        Args:
            log_entry: A log entry string

        Returns:
            Dictionary with tampering analysis
        """
        analysis = {
            'is_suspicious': False,
            'tampering_patterns': set(),
            'possible_original': '',
            'risk_level': 'low',
            'reasons': []
        }

        # Extract any domain information from the log entry
        domain_info = self.extract_domain_info(log_entry)

        if domain_info['has_domain']:
            # Check for suspicious TLDs
            if domain_info['tld'] in self.suspicious_tlds:
                analysis['is_suspicious'] = True
                analysis['tampering_patterns'].add('suspicious_tld')
                analysis['risk_level'] = 'medium'
                analysis['reasons'].append(f"Suspicious TLD found: .{domain_info['tld']}")
```

```
            # Check for subdomain issues
            if domain_info['subdomain']:
                subdomain = domain_info['subdomain']

                # Check for character substitutions
                for char in subdomain:
                    if char in self.char_substitutions:
                        analysis['is_suspicious'] = True
                        analysis['tampering_patterns'].add('character_substitution')
                        analysis['risk_level'] = 'high'
                        analysis['reasons'].append(f"Possible character substitution: '{char}'
might be '{self.char_substitutions[char]}'")

                        # Generate a possible original by replacing the character
                        possible_original = log_entry.replace(subdomain,
                                                              subdomain.replace(char,
self.char_substitutions[char]))
                        analysis['possible_original'] = possible_original

                # Check for similar but different subdomains
                for common_sub in self.common_subdomains:
                    if subdomain != common_sub and self.levenshtein_distance(subdomain,
common_sub) <= 2:
                        analysis['is_suspicious'] = True
                        analysis['tampering_patterns'].add('similar_subdomain')
                        analysis['risk_level'] = 'high'
                        analysis['reasons'].append(f"Subdomain '{subdomain}' is suspiciously
similar to common subdomain '{common_sub}'")

                        # Generate a possible original version
                        possible_original = log_entry.replace(subdomain, common_sub)
                        analysis['possible_original'] = possible_original

                # Check for unusually long subdomains (potential data exfiltration)
                if len(subdomain) > 30:
                    analysis['is_suspicious'] = True
                    analysis['tampering_patterns'].add('exfiltration_subdomain')
                    analysis['risk_level'] = 'high'
                    analysis['reasons'].append(f"Unusually long subdomain (length:
{len(subdomain)}) may indicate data exfiltration")
```

```
        # Check for IP address patterns
        ip_matches = re.findall(r'\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b', log_entry)
        if ip_matches:
            # Check for suspicious IP ranges
            for ip in ip_matches:
                octets = [int(octet) for octet in ip.split('.')]

                # Check for loopback or private IP misuse
                if octets[0] == 127 or (octets[0] == 10) or \
                   (octets[0] == 172 and 16 <= octets[1] <= 31) or \
                   (octets[0] == 192 and octets[1] == 168):
                    analysis['is_suspicious'] = True
                    analysis['tampering_patterns'].add('internal_ip_exposure')
                    analysis['risk_level'] = 'critical'
                    analysis['reasons'].append(f"Internal IP address exposed: {ip}")

        # Check for DNS record types and modifications
        record_types = ['A', 'AAAA', 'MX', 'CNAME', 'TXT', 'NS', 'SOA', 'SRV', 'PTR']
        for record_type in record_types:
            # Look for record type followed by manipulation indicators
            pattern = r'\b' + record_type +
r'\s+(?:changed|modified|updated|deleted|removed|added)\b'
            if re.search(pattern, log_entry, re.IGNORECASE):
                analysis['is_suspicious'] = True
                analysis['tampering_patterns'].add('dns_record_modification')
                analysis['risk_level'] = 'high'
                analysis['reasons'].append(f"DNS {record_type} record modification detected")

        # Look for DNS amplification or reflection attack patterns
        if re.search(r'\b(?:amplification|reflection|flood|ddos)\b', log_entry, re.IGNORECASE)
and domain_info['has_domain']:
            analysis['is_suspicious'] = True
            analysis['tampering_patterns'].add('dns_amplification')
            analysis['risk_level'] = 'critical'
            analysis['reasons'].append(f"Possible DNS amplification attack signature")

        # Update risk level based on number of patterns
        if len(analysis['tampering_patterns']) >= 3:
            analysis['risk_level'] = 'critical'
        elif len(analysis['tampering_patterns']) == 2:
```

```
            analysis['risk_level'] = 'high' if analysis['risk_level'] != 'critical' else
'critical'

        return analysis

    @staticmethod
    def levenshtein_distance(s1: str, s2: str) -> int:
        """
        Calculate the Levenshtein distance between two strings.

        Args:
            s1: First string
            s2: Second string

        Returns:
            The Levenshtein distance
        """
        if len(s1) < len(s2):
            return DNSSubdomainBatchChecker.levenshtein_distance(s2, s1)

        if len(s2) == 0:
            return len(s1)

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

        return previous_row[-1]

    def process_file_pair(self, message_file: str, hmac_file: str) -> Dict[str, Any]:
        """
        Process a single message/HMAC file pair.

        Args:
            message_file: Path to the message file
```

```
    hmac_file: Path to the HMAC file

Returns:
    Dictionary with processing results
"""
# Extract file number for identification
match = re.search(r'message_(\d+)\.txt$', message_file)
file_id = match.group(1) if match else os.path.basename(message_file)

# Read files
try:
    message_content = self.read_file(message_file)
    hmac_content = self.read_file(hmac_file)

    # Verify each line
    results = {
        'file_id': file_id,
        'message_file': message_file,
        'hmac_file': hmac_file,
        'total_lines': min(len(message_content), len(hmac_content)),
        'valid_lines': 0,
        'invalid_lines': 0,
        'suspicious_lines': 0,
        'invalid_entries': [],
        'tampering_summary': {
            'patterns': {},
            'risk_levels': {
                'low': 0,
                'medium': 0,
                'high': 0,
                'critical': 0
            }
        }
    }

    # Process lines
    for i in range(min(len(message_content), len(hmac_content))):
        message = message_content[i]
        signature = hmac_content[i]

        # Skip empty lines
```

```
                if not message or not signature:
                    continue

                # Verify HMAC
                is_valid = self.verify_hmac(message, signature)

                if is_valid:
                    results['valid_lines'] += 1
                else:
                    results['invalid_lines'] += 1

                    # Generate correct signature
                    correct_signature = self.calculate_hmac(message)

                    # Check for tampering
                    tampering_analysis = self.detect_tampering(message)

                    invalid_entry = {
                        'line_number': i + 1,
                        'message': message,
                        'provided_signature': signature,
                        'correct_signature': correct_signature,
                        'tampering_analysis': tampering_analysis
                    }

                    results['invalid_entries'].append(invalid_entry)

                    # Update tampering statistics
                    if tampering_analysis['is_suspicious']:
                        results['suspicious_lines'] += 1

results['tampering_summary']['risk_levels'][tampering_analysis['risk_level']] += 1

                        # Count pattern occurrences
                        for pattern in tampering_analysis['tampering_patterns']:
                            if pattern not in results['tampering_summary']['patterns']:
                                results['tampering_summary']['patterns'][pattern] = 0
                            results['tampering_summary']['patterns'][pattern] += 1

            return results
```

```
        except Exception as e:
            print(f"Error processing file pair ({message_file}, {hmac_file}): {e}")
            return {
                'file_id': file_id,
                'message_file': message_file,
                'hmac_file': hmac_file,
                'error': str(e)
            }

    def process_directory(self, directory: str) -> Dict[str, Any]:
        """
        Process all matching file pairs in a directory.

        Args:
            directory: Directory containing message_*.txt and message_*.hmac files

        Returns:
            Dictionary with processing results for all files
        """
        # Find all matching file pairs
        file_pairs = self.find_file_pairs(directory)

        if not file_pairs:
            print(f"No matching message/HMAC file pairs found in {directory}")
            return {'error': 'No matching file pairs found'}

        # Process each file pair
        results = {
            'directory': directory,
            'total_files': len(file_pairs),
            'processed_files': 0,
            'files_with_errors': 0,
            'total_lines_processed': 0,
            'total_invalid_lines': 0,
            'total_suspicious_lines': 0,
            'file_results': [],
            'overall_tampering_summary': {
                'patterns': {},
                'risk_levels': {
                    'low': 0,
                    'medium': 0,
```

```
                    'high': 0,
                    'critical': 0
                }
            }
        }

        for message_file, hmac_file in file_pairs:
            print(f"Processing file pair: {os.path.basename(message_file)} and
{os.path.basename(hmac_file)}")

            # Process file pair
            file_result = self.process_file_pair(message_file, hmac_file)
            results['file_results'].append(file_result)

            # Update overall statistics
            if 'error' in file_result:
                results['files_with_errors'] += 1
            else:
                results['processed_files'] += 1
                results['total_lines_processed'] += file_result['total_lines']
                results['total_invalid_lines'] += file_result['invalid_lines']
                results['total_suspicious_lines'] += file_result['suspicious_lines']

                # Aggregate tampering patterns
                for pattern, count in file_result['tampering_summary']['patterns'].items():
                    if pattern not in results['overall_tampering_summary']['patterns']:
                        results['overall_tampering_summary']['patterns'][pattern] = 0
                    results['overall_tampering_summary']['patterns'][pattern] += count

                # Aggregate risk levels
                for level in ['low', 'medium', 'high', 'critical']:
                    results['overall_tampering_summary']['risk_levels'][level] += \
                        file_result['tampering_summary']['risk_levels'][level]

        return results

    def save_corrected_hmac_files(self, results: Dict[str, Any], output_dir: str) -> None:
        """
        Save corrected HMAC files for each processed file pair.

        Args:
```

```
            results: Overall processing results
            output_dir: Output directory
        """
        corrected_dir = os.path.join(output_dir, 'corrected_hmac_files')
        os.makedirs(corrected_dir, exist_ok=True)

        for file_result in results['file_results']:
            if 'error' in file_result:
                continue

            # Get original file content
            message_file = file_result['message_file']
            hmac_file = file_result['hmac_file']

            try:
                # Read original message file
                message_content = self.read_file(message_file)

                # Create corrected HMAC file
                corrected_hmac_path = os.path.join(corrected_dir, os.path.basename(hmac_file))

                with open(corrected_hmac_path, 'w') as f:
                    for message in message_content:
                        if message:  # Skip empty lines
                            correct_signature = self.calculate_hmac(message)
                            f.write(f"{correct_signature}\n")

                print(f"Created corrected HMAC file: {corrected_hmac_path}")

            except Exception as e:
                print(f"Error creating corrected HMAC file for {os.path.basename(hmac_file)}:
{e}")

    def save_results(self, results: Dict[str, Any], output_dir: str) -> None:
        """
        Save processing results to output files.

        Args:
            results: Overall processing results
            output_dir: Output directory
        """
```

```
        os.makedirs(output_dir, exist_ok=True)

        # Save overall JSON results
        with open(os.path.join(output_dir, 'batch_results.json'), 'w') as f:
            # Convert sets to lists for JSON serialization
            serializable_results = json.dumps(results, indent=2, default=lambda x: list(x) if
isinstance(x, set) else x)
            f.write(serializable_results)

        # Save detailed report
        with open(os.path.join(output_dir, 'integrity_report.txt'), 'w') as f:
            f.write(f"DNS Subdomain Batch Integrity Report\n")
            f.write(f"==================================\n\n")
            f.write(f"Generated: {datetime.now().isoformat()}\n\n")

            f.write(f"Overall Summary:\n")
            f.write(f"---------------\n")
            f.write(f"Directory processed: {results['directory']}\n")
            f.write(f"Total file pairs: {results['total_files']}\n")
            f.write(f"Successfully processed: {results['processed_files']}\n")
            f.write(f"Files with errors: {results['files_with_errors']}\n")
            f.write(f"Total log lines processed: {results['total_lines_processed']}\n")
            f.write(f"Total invalid lines: {results['total_invalid_lines']}\n")
            f.write(f"Total suspicious lines: {results['total_suspicious_lines']}\n\n")

            # Risk level summary
            if results['total_suspicious_lines'] > 0:
                f.write(f"Risk Level Distribution:\n")
                for level in ['low', 'medium', 'high', 'critical']:
                    count = results['overall_tampering_summary']['risk_levels'][level]
                    indicator = '!' * (1 if level == 'low' else 2 if level == 'medium' else 3
if level == 'high' else 4)
                    f.write(f"  {indicator} {level.upper()}: {count}\n")

                f.write(f"\nTampering Patterns Detected:\n")
                for pattern, count in
sorted(results['overall_tampering_summary']['patterns'].items(),
                                           key=lambda x: x[1], reverse=True):
                    f.write(f"  - {pattern}: {count}\n")

            # Per-file summary
```

```
            f.write(f"\nPer-File Summary:\n")
            f.write(f"----------------\n")
            for file_result in results['file_results']:
                if 'error' in file_result:
                    f.write(f"File {file_result['file_id']}: ERROR -
{file_result['error']}\n")
                else:
                    integrity_status = "COMPROMISED" if file_result['invalid_lines'] > 0 else
"INTACT"
                    risk_level = "HIGH RISK" if
(file_result['tampering_summary']['risk_levels']['high'] > 0 or

file_result['tampering_summary']['risk_levels']['critical'] > 0) else \
                                 "MEDIUM RISK" if
file_result['tampering_summary']['risk_levels']['medium'] > 0 else \
                                 "LOW RISK" if file_result['suspicious_lines'] > 0 else "SAFE"

                    f.write(f"File {file_result['file_id']}: {integrity_status} -
{risk_level}\n")
                    f.write(f"  Message file:
{os.path.basename(file_result['message_file'])}\n")
                    f.write(f"  Lines: {file_result['total_lines']} total,
{file_result['invalid_lines']} invalid, {file_result['suspicious_lines']} suspicious\n")

                    if file_result['suspicious_lines'] > 0:
                        # Show the first few suspicious entries
                        suspicious_entries = [entry for entry in
file_result['invalid_entries']
                                              if entry['tampering_analysis']['is_suspicious']]

                        f.write(f"  Top suspicious entries ({min(3, len(suspicious_entries))}
of {len(suspicious_entries)}):\n")
                        for i, entry in enumerate(suspicious_entries[:3]):
                            f.write(f"    Line {entry['line_number']}:
{entry['message'][:50]}{'...' if len(entry['message']) > 50 else ''}\n")
                            f.write(f"      Risk:
{entry['tampering_analysis']['risk_level'].upper()}\n")
                            f.write(f"      Patterns: {',
'.join(entry['tampering_analysis']['tampering_patterns'])}\n")

                    f.write("\n")
```

```
        # Create a file with high-risk entries for immediate attention
        high_risk_entries = []
        for file_result in results['file_results']:
            if 'error' in file_result:
                continue

            file_id = file_result['file_id']
            for entry in file_result['invalid_entries']:
                if entry['tampering_analysis']['is_suspicious'] and \
                   entry['tampering_analysis']['risk_level'] in ['high', 'critical']:
                    entry_copy = entry.copy()
                    entry_copy['file_id'] = file_id
                    high_risk_entries.append(entry_copy)

        if high_risk_entries:
            with open(os.path.join(output_dir, 'high_risk_entries.txt'), 'w') as f:
                f.write(f"HIGH RISK DNS LOG ENTRIES - IMMEDIATE ATTENTION REQUIRED\n")
                f.write(f"==================================================\n\n")
                f.write(f"Generated: {datetime.now().isoformat()}\n")
                f.write(f"Total high-risk entries: {len(high_risk_entries)}\n\n")

                # Sort by risk level (critical first)
                high_risk_entries.sort(key=lambda x: 0 if
x['tampering_analysis']['risk_level'] == 'critical' else 1)

                for entry in high_risk_entries:
                    f.write(f"File {entry['file_id']}, Line {entry['line_number']} -
[{entry['tampering_analysis']['risk_level'].upper()}]\n")
                    f.write(f"  Message: {entry['message']}\n")
                    f.write(f"  Provided signature: {entry['provided_signature']}\n")
                    f.write(f"  Correct signature: {entry['correct_signature']}\n")
                    f.write(f"  Tampering patterns: {',
'.join(entry['tampering_analysis']['tampering_patterns'])}\n")
                    f.write(f"  Reasons:\n")
                    for reason in entry['tampering_analysis']['reasons']:
                        f.write(f"    - {reason}\n")

                    if entry['tampering_analysis']['possible_original']:
                        f.write(f"  Possible original:
{entry['tampering_analysis']['possible_original']}\n")
```

```
                    f.write("\n")

        # Save corrected HMAC files
        self.save_corrected_hmac_files(results, output_dir)

def main():
    """Main entry point for the script."""
    parser = argparse.ArgumentParser(description='DNS Subdomain Batch Integrity Checker')
    parser.add_argument('--directory', '-d', required=True, help='Directory containing log
files')
    parser.add_argument('--output', '-o', default='batch_output', help='Output directory
(default: batch_output)')
    parser.add_argument('--key', '-k', default=VALID_KEY, help=f'HMAC signing key (default:
{VALID_KEY})')

    args = parser.parse_args()

    checker = DNSSubdomainBatchChecker(key=args.key)

    try:
        start_time = datetime.now()
        print(f"Starting batch processing of DNS log files in {args.directory}")
        print(f"Started at: {start_time.isoformat()}")

        results = checker.process_directory(args.directory)

        if 'error' in results:
            print(f"Error: {results['error']}")
            sys.exit(1)

        # Save results
        checker.save_results(results, args.output)

        end_time = datetime.now()
        duration = end_time - start_time

        print(f"\nBatch processing completed!")
        print(f"Duration: {duration.total_seconds():.2f} seconds")
        print(f"Files processed: {results['processed_files']} of {results['total_files']}")
        print(f"Total lines checked: {results['total_lines_processed']}")
```

```
        print(f"Invalid lines detected: {results['total_invalid_lines']}")
        print(f"Suspicious lines detected: {results['total_suspicious_lines']}")
        print(f"Results saved to: {args.output}")

        if results['total_suspicious_lines'] > 0:
            print(f"\n⚠ WARNING: {results['total_suspicious_lines']} suspicious log entries
detected!")
            high_risk = results['overall_tampering_summary']['risk_levels']['high'] + \
                        results['overall_tampering_summary']['risk_levels']['critical']

            if high_risk > 0:
                print(f"🚨 CRITICAL: {high_risk} high or critical risk entries found!")
                print(f"Check {os.path.join(args.output, 'high_risk_entries.txt')} for
details")

    except Exception as e:
        print(f"Error: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)

if __name__ == "__main__":
    main()
```

Revision #1

Created 2025-11-25 18:05:17 UTC by David Rizzo

Updated 2025-11-25 18:05:41 UTC by David Rizzo