Incident Forensics and Analysis - Complete Implementation Guide

1. Digital Forensics Framework

1.1 Core Forensics Engine

```python
```

```python
# forensics_engine.py - Main digital forensics analysis engine
import os
import hashlib
import json
import struct
import mmap
import sqlite3
from datetime import datetime
from typing import Dict, List, Optional, Tuple, Any
import volatility3
import yara
import magic
import pyewf
import pytsk3

class DigitalForensicsEngine:
    """Comprehensive digital forensics analysis system"""

    def __init__(self):
        self.evidence_store = EvidenceStore()
        self.chain_of_custody = ChainOfCustody()
        self.analysis_modules = {
            'memory': MemoryForensics(),
            'disk': DiskForensics(),
            'network': NetworkForensics(),
            'malware': MalwareAnalysis(),
            'mobile': MobileForensics(),
            'cloud': CloudForensics()
        }

    async def process_evidence(
        self,
        evidence_source: str,
        evidence_type: str,
        case_id: str
    ) -> Dict:
        """Process digital evidence with chain of custody"""

        # Create evidence record
        evidence_id = self.chain_of_custody.create_record({
            'source': evidence_source,
            'type': evidence_type,
            'case_id': case_id,
            'collected_at': datetime.utcnow(),
            'collector': self.get_current_analyst()
        })

        try:
            # Acquire evidence
            evidence_data = await self.acquire_evidence(
                evidence_source,
                evidence_type
            )

            # Calculate hashes for integrity
            hashes = self.calculate_hashes(evidence_data)
            self.chain_of_custody.add_hashes(evidence_id, hashes)

            # Store evidence securely
            storage_path = await self.evidence_store.store(
                evidence_data,
                evidence_id
            )

            # Perform initial triage
            triage_results = await self.triage_evidence(
                evidence_data,
                evidence_type
            )

            # Deep analysis based on type
            analysis_results = await self.analyze_evidence(
                evidence_data,
                evidence_type,
                triage_results
            )

            # Generate report
            report = await self.generate_forensic_report(
                evidence_id,
                analysis_results
            )

            return {
                'evidence_id': evidence_id,
                'storage_path': storage_path,
                'hashes': hashes,
                'triage': triage_results,
                'analysis': analysis_results,
                'report': report
            }

        except Exception as e:
            self.chain_of_custody.mark_error(evidence_id, str(e))
            raise

    def calculate_hashes(self, data: bytes) -> Dict[str, str]:
        """Calculate multiple hashes for evidence integrity"""

        return {
            'md5': hashlib.md5(data).hexdigest(),
            'sha1': hashlib.sha1(data).hexdigest(),
            'sha256': hashlib.sha256(data).hexdigest(),
            'sha512': hashlib.sha512(data).hexdigest(),
            'ssdeep': self.calculate_fuzzy_hash(data)
        }
```

1.2 Memory Forensics

python

```python
class MemoryForensics:
    """Advanced memory forensics and analysis"""

    def __init__(self):
        self.volatility_framework = self.init_volatility()
        self.yara_rules = self.load_yara_rules()

    async def analyze_memory_dump(
        self,
        memory_dump: str,
        os_profile: str = None
    ) -> Dict:
        """Comprehensive memory dump analysis"""

        # Auto-detect OS profile if not provided
        if not os_profile:
            os_profile = await self.detect_profile(memory_dump)

        analysis_results = {
            'profile': os_profile,
            'processes': await self.extract_processes(memory_dump),
            'network_connections': await self.extract_network(memory_dump),
            'registry_keys': await self.extract_registry(memory_dump),
            'loaded_drivers': await self.extract_drivers(memory_dump),
            'hooks': await self.detect_hooks(memory_dump),
            'injected_code': await self.detect_injections(memory_dump),
            'hidden_processes': await self.find_hidden_processes(memory_dump),
            'malware_artifacts': await self.scan_for_malware(memory_dump),
            'strings': await self.extract_strings(memory_dump),
            'timeline': await self.create_timeline(memory_dump)
        }

        return analysis_results

    async def extract_processes(self, memory_dump: str) -> List[Dict]:
        """Extract and analyze running processes"""

        processes = []

        # Use Volatility to extract process list
        process_list = self.volatility_framework.pslist(memory_dump)

        for proc in process_list:
            process_info = {
                'pid': proc.pid,
                'ppid': proc.ppid,
                'name': proc.name,
                'path': proc.path,
                'cmdline': proc.cmdline,
                'create_time': proc.create_time,
                'threads': proc.num_threads,
                'handles': proc.num_handles,
                'vads': await self.analyze_vads(memory_dump, proc.pid),
                'dlls': await self.extract_dlls(memory_dump, proc.pid),
                'suspicious': await self.check_suspicious_process(proc)
            }

            processes.append(process_info)

        return processes

    async def detect_injections(self, memory_dump: str) -> List[Dict]:
        """Detect process injection techniques"""

        injections = []

        # Check for various injection techniques
        techniques = [
            self.detect_process_hollowing,
            self.detect_reflective_dll,
            self.detect_atom_bombing,
            self.detect_setwindowshook,
            self.detect_apc_injection
        ]

        for technique in techniques:
            results = await technique(memory_dump)
            if results:
                injections.extend(results)

        return injections

    async def detect_process_hollowing(self, memory_dump: str) -> List[Dict]:
        """Detect process hollowing technique"""

        hollowed = []
        processes = self.volatility_framework.pslist(memory_dump)

        for proc in processes:
            # Check if process image doesn't match memory
            vad_info = await self.analyze_vads(memory_dump, proc.pid)

            for vad in vad_info:
                if vad['protection'] == 'PAGE_EXECUTE_READWRITE':
                    # Check if VAD contains PE header
                    if await self.contains_pe_header(vad['data']):
                        if not await self.matches_disk_image(proc.path, vad['data']):
                            hollowed.append({
                                'type': 'process_hollowing',
                                'pid': proc.pid,
                                'process': proc.name,
                                'vad_address': vad['start'],
                                'evidence': 'PE in memory doesn\'t match disk'
                            })

        return hollowed
```

1.3 Disk Forensics

```python
```

```
class DiskForensics:
    """Disk image analysis and file system forensics"""

    def __init__(self):
        self.file_signatures = self.load_file_signatures()
        self.carving_patterns = self.load_carving_patterns()

    async def analyze_disk_image(
        self,
        image_path: str,
        image_type: str = 'raw'
    ) -> Dict:
        """Comprehensive disk image analysis"""

        # Open disk image
        if image_type == 'e01':
            img_info = pyewf.handle()
            img_info.open(image_path)
        else:
            img_info = pytsk3.Img_Info(image_path)

        # Analyze file system
        fs_info = pytsk3.FS_Info(img_info)

        analysis = {
            'partition_table': await self.analyze_partitions(img_info),
            'file_system': fs_info.info.ftype,
            'deleted_files': await self.recover_deleted_files(fs_info),
            'timeline': await self.create_timeline(fs_info),
            'carved_files': await self.carve_files(img_info),
            'registry_analysis': await self.analyze_registry(fs_info),
            'browser_artifacts': await self.extract_browser_data(fs_info),
            'usb_history': await self.extract_usb_history(fs_info),
            'prefetch_analysis': await self.analyze_prefetch(fs_info),
            'event_logs': await self.parse_event_logs(fs_info),
            'shadow_copies': await self.analyze_shadow_copies(img_info)
        }

        return analysis

    async def recover_deleted_files(
        self,
        fs_info: pytsk3.FS_Info
    ) -> List[Dict]:
        """Recover deleted files from file system"""

        deleted_files = []

        # Walk through file system
        directory = fs_info.open_dir("/")

        for entry in self.walk_filesystem(directory):
            if entry.info.meta and \
               entry.info.meta.flags == pytsk3.TSK_FS_META_FLAG_UNALLOC:

                # Attempt to recover file
                file_data = self.read_file_slack(entry)

                if file_data:
                    recovered = {
                        'name': entry.info.name.name.decode('utf-8', errors='replace'),
                        'size': entry.info.meta.size,
                        'deleted_time': entry.info.meta.dtime,
                        'recovered_data': file_data,
                        'recovery_confidence': self.assess_recovery_confidence(file_data)
                    }
                    deleted_files.append(recovered)

        return deleted_files

    async def carve_files(self, img_info) -> List[Dict]:
        """Carve files from unallocated space"""
        carved_files = []
        block_size = 512

        # Read image in blocks
        offset = 0
        img_size = img_info.get_size()

        while offset < img_size:
            # Read block
            data = img_info.read(offset, block_size)

            # Check for file signatures
            for sig_name, signature in self.file_signatures.items():
                if data.startswith(signature['header']):
                    # Found potential file start
                    carved_file = await self.carve_single_file(
                        img_info,
                        offset,
                        signature
                    )

                    if carved_file:
                        carved_files.append({
                            'type': sig_name,
                            'offset': offset,
                            'size': len(carved_file),
                            'data': carved_file,
                            'hash': hashlib.sha256(carved_file).hexdigest()
                        })

            offset += block_size

        return carved_files
```

1.4 Network Forensics

```python
```

```python
class NetworkForensics:
    """Network traffic analysis and forensics"""

    def __init__(self):
        self.protocol_parsers = {
            'http': HTTPParser(),
            'https': TLSParser(),
            'dns': DNSParser(),
            'smtp': SMTPParser(),
            'ftp': FTPParser()
        }

    async def analyze_pcap(self, pcap_file: str) -> Dict:
        """Analyze network capture file"""

        import pyshark

        capture = pyshark.FileCapture(pcap_file)

        analysis = {
            'statistics': await self.calculate_statistics(capture),
            'conversations': await self.extract_conversations(capture),
            'dns_queries': await self.extract_dns(capture),
            'http_sessions': await self.reconstruct_http(capture),
            'file_transfers': await self.extract_files(capture),
            'credentials': await self.extract_credentials(capture),
            'anomalies': await self.detect_anomalies(capture),
            'ioc_matches': await self.match_network_iocs(capture),
            'timeline': await self.create_network_timeline(capture)
        }

        return analysis

    async def reconstruct_http(self, capture) -> List[Dict]:
        """Reconstruct HTTP sessions from packet capture"""

        http_sessions = []
        current_sessions = {}

        for packet in capture:
            if 'HTTP' in packet:
                # Extract HTTP data
                if hasattr(packet.http, 'request'):
                    # HTTP Request
                    session_key = f"{packet.ip.src}:{packet.tcp.srcport}->{packet.ip.dst}:{packet.tcp.dstport}"

                    current_sessions[session_key] = {
                        'timestamp': packet.sniff_timestamp,
                        'method': packet.http.request_method,
                        'uri': packet.http.request_uri,
                        'host': packet.http.host if hasattr(packet.http, 'host') else '',
                        'user_agent': packet.http.user_agent if hasattr(packet.http, 'user_agent') else '',
                        'request_headers': self.extract_headers(packet.http),
                        'request_body': await self.extract_http_body(packet)
                    }

                elif hasattr(packet.http, 'response'):
                    # HTTP Response
                    session_key = f"{packet.ip.dst}:{packet.tcp.dstport}->{packet.ip.src}:{packet.tcp.srcport}"

                    if session_key in current_sessions:
                        session = current_sessions[session_key]
                        session['response_code'] = packet.http.response_code
                        session['response_headers'] = self.extract_headers(packet.http)
                        session['response_body'] = await self.extract_http_body(packet)

                        http_sessions.append(session)
                        del current_sessions[session_key]

        return http_sessions

    async def extract_files(self, capture) -> List[Dict]:
        """Extract transferred files from network capture"""
        extracted_files = []
        file_streams = {}

        for packet in capture:
            # Check for file transfer protocols
            if 'FTP-DATA' in packet:
                await self.extract_ftp_file(packet, file_streams)
            elif 'SMB' in packet:
                await self.extract_smb_file(packet, file_streams)
            elif 'HTTP' in packet and hasattr(packet.http, 'file_data'):
                await self.extract_http_file(packet, file_streams)

        # Reassemble file streams
        for stream_id, stream_data in file_streams.items():
            file_data = b''.join(stream_data['chunks'])

            extracted_files.append({
                'protocol': stream_data['protocol'],
                'filename': stream_data.get('filename', 'unknown'),
                'size': len(file_data),
                'hash': hashlib.sha256(file_data).hexdigest(),
                'mime_type': magic.from_buffer(file_data, mime=True),
                'data': file_data
            })

        return extracted_files
```

1.5 Malware Analysis

```python
```

```python
class MalwareAnalysis:
    """Advanced malware analysis and reverse engineering"""

    def __init__(self):
        self.sandbox = SandboxEnvironment()
        self.static_analyzer = StaticAnalyzer()
        self.dynamic_analyzer = DynamicAnalyzer()

    async def analyze_malware(
        self,
        sample_path: str,
        analysis_depth: str = 'deep'
    ) -> Dict:
        """Comprehensive malware analysis"""

        # Calculate hashes
        with open(sample_path, 'rb') as f:
            sample_data = f.read()

        sample_hashes = {
            'md5': hashlib.md5(sample_data).hexdigest(),
            'sha1': hashlib.sha1(sample_data).hexdigest(),
            'sha256': hashlib.sha256(sample_data).hexdigest(),
            'ssdeep': self.calculate_fuzzy_hash(sample_data),
            'imphash': await self.calculate_imphash(sample_data)
        }

        analysis_results = {
            'hashes': sample_hashes,
            'file_info': await self.get_file_info(sample_path),
            'static_analysis': await self.static_analysis(sample_path),
            'dynamic_analysis': await self.dynamic_analysis(sample_path),
            'behavioral_analysis': await self.behavioral_analysis(sample_path),
            'network_analysis': await self.network_behavior_analysis(sample_path),
            'code_analysis': await self.code_analysis(sample_path),
            'yara_matches': await self.yara_scan(sample_data),
            'unpacked_samples': await self.unpack_sample(sample_data),
            'ioc_extraction': await self.extract_iocs(sample_path),
            'classification': await self.classify_malware(analysis_results)
        }

        return analysis_results

    async def static_analysis(self, sample_path: str) -> Dict:
        """Perform static malware analysis"""

        import pefile
        import capstone

        static_results = {}

        try:
            # Parse PE file
            pe = pefile.PE(sample_path)

            # Extract PE information
            static_results['pe_info'] = {
                'machine': hex(pe.FILE_HEADER.Machine),
                'timestamp': datetime.fromtimestamp(pe.FILE_HEADER.TimeDateStamp),
                'sections': self.analyze_sections(pe),
                'imports': self.analyze_imports(pe),
                'exports': self.analyze_exports(pe),
                'resources': self.analyze_resources(pe),
                'version_info': self.extract_version_info(pe),
                'signatures': self.check_signatures(pe),
                'entropy': self.calculate_entropy(pe),
                'packed': self.check_if_packed(pe)
            }

            # Disassemble code
            static_results['disassembly'] = await self.disassemble_code(pe)

            # String analysis
            static_results['strings'] = await self.extract_strings(sample_path)

            # Anti-analysis techniques
            static_results['anti_analysis'] = await self.detect_anti_analysis(pe)

        except Exception as e:
            static_results['error'] = str(e)

        return static_results

    async def dynamic_analysis(self, sample_path: str) -> Dict:
        """Perform dynamic malware analysis in sandbox"""

        # Prepare sandbox environment
        sandbox_id = await self.sandbox.prepare_environment()

        try:
            # Execute sample in sandbox
            execution_results = await self.sandbox.execute_sample(
                sample_path,
                sandbox_id,
                timeout=300  # 5 minutes
            )

            # Collect artifacts
            artifacts = await self.sandbox.collect_artifacts(sandbox_id)

            dynamic_results = {
                'processes_created': artifacts['processes'],
                'files_created': artifacts['files_created'],
                'files_modified': artifacts['files_modified'],
                'files_deleted': artifacts['files_deleted'],
                'registry_modifications': artifacts['registry'],
                'network_connections': artifacts['network'],
                'dns_queries': artifacts['dns'],
                'api_calls': artifacts['api_calls'],
                'mutex_created': artifacts['mutex'],
                'services_created': artifacts['services'],
                'dropped_files': await self.analyze_dropped_files(artifacts),
                'memory_artifacts': await self.analyze_memory_artifacts(sandbox_id),
                'screenshots': artifacts['screenshots']
            }

            return dynamic_results

        finally:
            # Clean up sandbox
            await self.sandbox.cleanup(sandbox_id)
```

1.6 Timeline Analysis

```python
```

```
class TimelineAnalysis:
    """Forensic timeline creation and analysis"""

    def __init__(self):
        self.timeline_sources = []
        self.events = []

    async def create_super_timeline(
        self,
        evidence_sources: List[Dict]
    ) -> Dict:
        """Create comprehensive forensic timeline"""

        timeline_events = []

        for source in evidence_sources:
            if source['type'] == 'disk_image':
                events = await self.extract_disk_timeline(source['path'])
                timeline_events.extend(events)

            elif source['type'] == 'memory_dump':
                events = await self.extract_memory_timeline(source['path'])
                timeline_events.extend(events)

            elif source['type'] == 'event_logs':
                events = await self.parse_event_logs(source['path'])
                timeline_events.extend(events)

            elif source['type'] == 'network_capture':
                events = await self.extract_network_timeline(source['path'])
                timeline_events.extend(events)

        # Sort and correlate events
        sorted_timeline = sorted(timeline_events, key=lambda x: x['timestamp'])

        # Identify patterns and anomalies
        patterns = await self.identify_patterns(sorted_timeline)
        anomalies = await self.detect_timeline_anomalies(sorted_timeline)

        # Create activity clusters
        clusters = await self.cluster_activities(sorted_timeline)

        return {
            'events': sorted_timeline,
            'total_events': len(sorted_timeline),
            'time_range': {
                'start': sorted_timeline[0]['timestamp'] if sorted_timeline else None,
                'end': sorted_timeline[-1]['timestamp'] if sorted_timeline else None
            },
            'patterns': patterns,
            'anomalies': anomalies,
            'activity_clusters': clusters,
            'critical_events': await self.identify_critical_events(sorted_timeline)
        }

    async def extract_disk_timeline(self, disk_image: str) -> List[Dict]:
        """Extract timeline from disk image"""

        events = []

        # File system timeline
        fs_timeline = await self.extract_filesystem_timeline(disk_image)
        events.extend(fs_timeline)

        # Registry timeline
        reg_timeline = await self.extract_registry_timeline(disk_image)
        events.extend(reg_timeline)

        # Browser history timeline
        browser_timeline = await self.extract_browser_timeline(disk_image)
        events.extend(browser_timeline)

        # Application logs timeline
        app_timeline = await self.extract_application_timeline(disk_image)
        events.extend(app_timeline)

        return events
```

2. Incident Analysis Tools

2.1 Root Cause Analysis

```python
```

```
class RootCauseAnalyzer:
    """Automated root cause analysis for security incidents"""

    def __init__(self):
        self.causal_models = {}
        self.attack_patterns = self.load_attack_patterns()

    async def analyze_incident(
        self,
        incident_data: Dict,
        evidence: List[Dict]
    ) -> Dict:
        """Perform root cause analysis on security incident"""

        # Build incident graph
        incident_graph = await self.build_incident_graph(
            incident_data,
            evidence
        )

        # Identify attack vectors
        attack_vectors = await self.identify_attack_vectors(
            incident_graph
        )

        # Trace attack path
        attack_path = await self.trace_attack_path(
            incident_graph,
            attack_vectors
        )

        # Identify root cause
        root_causes = await self.identify_root_causes(
            attack_path,
            incident_data
        )

        # Generate remediation recommendations
        recommendations = await self.generate_recommendations(
            root_causes
        )

        return {
            'incident_id': incident_data['id'],
            'attack_vectors': attack_vectors,
            'attack_path': attack_path,
            'root_causes': root_causes,
            'contributing_factors': await self.identify_contributing_factors(incident_graph),
            'timeline': await self.reconstruct_attack_timeline(incident_graph),
            'impact_analysis': await self.analyze_impact(incident_data),
            'recommendations': recommendations,
            'lessons_learned': await self.extract_lessons_learned(incident_data)
        }

    async def build_incident_graph(
        self,
        incident_data: Dict,
        evidence: List[Dict]
    ) -> Dict:
        """Build causal graph of incident"""

        import networkx as nx

        # Create directed graph
        graph = nx.DiGraph()

        # Add nodes for each event/artifact
        for item in evidence:
            graph.add_node(
                item['id'],
                type=item['type'],
                timestamp=item['timestamp'],
                data=item['data']
    )

# Add edges based on causal relationships
for i, item1 in enumerate(evidence):
    for item2 in evidence[i+1:]:
        if self.is_causally_related(item1, item2):
            graph.add_edge(
                item1['id'],
                item2['id'],
                relationship=self.determine_relationship(item1, item2)
            )

# Identify critical paths
critical_paths = nx.all_simple_paths(
    graph,
    source=self.find_initial_compromise(graph),
    target=self.find_impact_point(graph)
)

return {
    'graph': graph,
    'critical_paths': list(critical_paths),
    'node_count': graph.number_of_nodes(),
    'edge_count': graph.number_of_edges()
}
```

2.2 Attack Reconstruction

python

```python
class AttackReconstructor:
    """Reconstruct attack scenarios from forensic evidence"""

    def __init__(self):
        self.mitre_framework = MITREATTACKFramework()
        self.attack_patterns = {}

    async def reconstruct_attack(
        self,
        evidence: List[Dict]
    ) -> Dict:
        """Reconstruct complete attack scenario"""

        # Phase 1: Initial Access
        initial_access = await self.identify_initial_access(evidence)

        # Phase 2: Execution
        execution = await self.identify_execution_methods(evidence)

        # Phase 3: Persistence
        persistence = await self.identify_persistence_mechanisms(evidence)

        # Phase 4: Privilege Escalation
        priv_escalation = await self.identify_privilege_escalation(evidence)

        # Phase 5: Defense Evasion
        defense_evasion = await self.identify_evasion_techniques(evidence)

        # Phase 6: Credential Access
        credential_access = await self.identify_credential_theft(evidence)

        # Phase 7: Discovery
        discovery = await self.identify_discovery_actions(evidence)

        # Phase 8: Lateral Movement
        lateral_movement = await self.identify_lateral_movement(evidence)

        # Phase 9: Collection
        collection = await self.identify_data_collection(evidence)

        # Phase 10: Exfiltration
        exfiltration = await self.identify_exfiltration(evidence)

        # Phase 11: Impact
        impact = await self.identify_impact(evidence)

        # Map to MITRE ATT&CK
        mitre_mapping = await self.map_to_mitre_attack({
            'initial_access': initial_access,
            'execution': execution,
            'persistence': persistence,
            'privilege_escalation': priv_escalation,
            'defense_evasion': defense_evasion,
            'credential_access': credential_access,
            'discovery': discovery,
            'lateral_movement': lateral_movement,
            'collection': collection,
            'exfiltration': exfiltration,
            'impact': impact
        })

        return {
            'kill_chain': {
                'initial_access': initial_access,
                'execution': execution,
                'persistence': persistence,
                'privilege_escalation': priv_escalation,
                'defense_evasion': defense_evasion,
                'credential_access': credential_access,
                'discovery': discovery,
                'lateral_movement': lateral_movement,
                'collection': collection,
                'exfiltration': exfiltration,
                'impact': impact
            },
            'mitre_attack_mapping': mitre_mapping,
            'attack_timeline': await self.create_attack_timeline(evidence),
            'attacker_profile': await self.profile_attacker(evidence),
            'confidence_score': await self.calculate_confidence(evidence)
        }
```

3. Evidence Collection & Preservation

3.1 Live System Forensics

```python
```

```python
class LiveForensics:
    """Live system evidence collection"""

    def __init__(self):
        self.collection_tools = {}
        self.evidence_container = EvidenceContainer()

    async def collect_live_evidence(
        self,
        target_system: str,
        collection_profile: str = 'comprehensive'
    ) -> Dict:
        """Collect evidence from live system"""

        evidence_package = {
            'system': target_system,
            'collection_time': datetime.utcnow(),
            'collector': self.get_current_analyst(),
            'artifacts': {}
        }

        # System information
        evidence_package['artifacts']['system_info'] = await self.collect_system_info(target_system)

        # Memory acquisition
        evidence_package['artifacts']['memory'] = await self.acquire_memory(target_system)

        # Running processes
        evidence_package['artifacts']['processes'] = await self.collect_processes(target_system)

        # Network connections
        evidence_package['artifacts']['network'] = await self.collect_network_state(target_system)

        # Open files and handles
        evidence_package['artifacts']['handles'] = await self.collect_handles(target_system)

        # Registry (Windows)
        if await self.is_windows(target_system):
            evidence_package['artifacts']['registry'] = await self.collect_registry(target_system)

        # Log files
        evidence_package['artifacts']['logs'] = await self.collect_logs(target_system)

        # User artifacts
        evidence_package['artifacts']['user_data'] = await self.collect_user_artifacts(target_system)

        # Create forensic image
        forensic_image = await self.create_forensic_package(evidence_package)

        return forensic_image

    async def acquire_memory(self, target: str) -> Dict:
        """Acquire memory from live system"""

        memory_acquisition = {
            'method': 'live_acquisition',
            'tool': 'winpmem',  # or appropriate tool
            'timestamp': datetime.utcnow()
        }

        # Execute memory acquisition
        if await self.is_windows(target):
            dump_path = await self.acquire_windows_memory(target)
        elif await self.is_linux(target):
            dump_path = await self.acquire_linux_memory(target)
        elif await self.is_macos(target):
            dump_path = await self.acquire_macos_memory(target)

        # Calculate hash of memory dump
        dump_hash = await self.calculate_file_hash(dump_path)

        memory_acquisition['dump_path'] = dump_path
        memory_acquisition['hash'] = dump_hash
        memory_acquisition['size'] = os.path.getsize(dump_path)

        return memory_acquisition
```

3.2 Chain of Custody Management

```python
```

```
class ChainOfCustody:
    """Maintain chain of custody for digital evidence"""

    def __init__(self):
        self.custody_db = sqlite3.connect('chain_of_custody.db')
        self.init_database()

    def create_evidence_record(
        self,
        evidence_data: Dict
    ) -> str:
        """Create new evidence record with unique ID"""

        evidence_id = self.generate_evidence_id()

        # Create custody record
        custody_record = {
            'evidence_id': evidence_id,
            'case_id': evidence_data['case_id'],
            'description': evidence_data['description'],
            'source': evidence_data['source'],
            'collection_method': evidence_data['collection_method'],
            'collector_name': evidence_data['collector'],
            'collector_badge': evidence_data.get('badge_number'),
            'collection_date': datetime.utcnow(),
            'initial_hash': evidence_data['hash'],
            'location': evidence_data['storage_location'],
            'status': 'collected',
            'seal_number': self.generate_seal_number()
        }

        # Store in database
        self.store_custody_record(custody_record)

        # Generate custody form
        custody_form = self.generate_custody_form(custody_record)

        return evidence_id

    def transfer_custody(
        self,
        evidence_id: str,
        from_person: str,
        to_person: str,
        reason: str
    ) -> bool:
        """Record custody transfer"""

        transfer_record = {
            'evidence_id': evidence_id,
            'from_person': from_person,
            'to_person': to_person,
            'transfer_date': datetime.utcnow(),
            'reason': reason,
            'from_signature': self.generate_digital_signature(from_person),
            'to_signature': self.generate_digital_signature(to_person)
        }

        # Verify evidence integrity
        if not self.verify_evidence_integrity(evidence_id):
            raise IntegrityError("Evidence integrity check failed")

        # Record transfer
        self.record_transfer(transfer_record)

        return True
```

4. Forensic Reporting

4.1 Automated Report Generation

python

```
class ForensicReportGenerator:
    """Generate comprehensive forensic reports"""

    def __init__(self):
        self.report_templates = {}
        self.visualization_engine = VisualizationEngine()

    async def generate_report(
        self,
        case_id: str,
        evidence: List[Dict],
        analysis_results: Dict,
        report_type: str = 'executive'
    ) -> Dict:
        """Generate forensic investigation report"""

        report = {
            'case_id': case_id,
            'report_id': self.generate_report_id(),
            'generated_date': datetime.utcnow(),
            'report_type': report_type,
            'sections': {}
        }

        # Executive Summary
        report['sections']['executive_summary'] = await self.create_executive_summary(
            case_id,
            analysis_results
        )

        # Investigation Overview
        report['sections']['investigation_overview'] = {
            'scope': analysis_results.get('scope'),
            'methodology': analysis_results.get('methodology'),
            'tools_used': analysis_results.get('tools'),
            'timeline': analysis_results.get('timeline')
        }

        # Evidence Summary
        report['sections']['evidence_summary'] = await self.summarize_evidence(evidence)

        # Technical Findings
        report['sections']['technical_findings'] = await self.format_technical_findings(
            analysis_results
        )

        # Attack Reconstruction
        report['sections']['attack_reconstruction'] = await self.format_attack_reconstruction(
            analysis_results.get('attack_reconstruction')
        )

        # Indicators of Compromise
        report['sections']['iocs'] = await self.format_iocs(
            analysis_results.get('indicators')
        )

        # Timeline Analysis
        report['sections']['timeline'] = await self.format_timeline(
            analysis_results.get('timeline')
        )

        # Visualizations
        report['sections']['visualizations'] = await self.create_visualizations(
            analysis_results
        )

        # Recommendations
        report['sections']['recommendations'] = await self.generate_recommendations(
            analysis_results
        )

        # Appendices
        report['sections']['appendices'] = await self.create_appendices(
            evidence,
            analysis_results
        )

        # Generate different formats
        report_formats = {
            'html': await self.generate_html_report(report),
            'pdf': await self.generate_pdf_report(report),
            'json': report,
            'markdown': await self.generate_markdown_report(report)
        }

        return report_formats
```

5. Advanced Analysis Techniques

5.1 Machine Learning for Forensics

```python
```

```python
class ForensicsML:
    """Machine learning for forensic analysis"""

    def __init__(self):
        self.models = {
            'anomaly_detection': self.load_anomaly_model(),
            'malware_classification': self.load_malware_model(),
            'user_behavior': self.load_behavior_model(),
            'data_exfiltration': self.load_exfiltration_model()
        }

    async def detect_anomalies(
        self,
        timeline_data: List[Dict]
    ) -> List[Dict]:
        """Detect anomalies in forensic timeline"""

        # Prepare features
        features = self.extract_timeline_features(timeline_data)

        # Apply isolation forest
        from sklearn.ensemble import IsolationForest

        clf = IsolationForest(contamination=0.1)
        predictions = clf.fit_predict(features)

        anomalies = []
        for i, pred in enumerate(predictions):
            if pred == -1:  # Anomaly
                anomaly = {
                    'event': timeline_data[i],
                    'anomaly_score': clf.score_samples([features[i]])[0],
                    'reason': await self.explain_anomaly(timeline_data[i], features[i])
                }
                anomalies.append(anomaly)

        return anomalies

    async def classify_behavior(
        self,
        user_actions: List[Dict]
    ) -> Dict:
        """Classify user behavior patterns"""

        # Extract behavioral features
        features = self.extract_behavior_features(user_actions)

        # Apply clustering
        from sklearn.cluster import DBSCAN

        clustering = DBSCAN(eps=0.3, min_samples=10)
        clusters = clustering.fit_predict(features)

        # Analyze clusters
        behavior_patterns = {}
        for cluster_id in set(clusters):
            if cluster_id != -1:  # Not noise
                cluster_actions = [
                    user_actions[i] for i, c in enumerate(clusters) if c == cluster_id
                ]

                pattern = await self.analyze_behavior_pattern(cluster_actions)
                behavior_patterns[f'pattern_{cluster_id}'] = pattern

        return behavior_patterns
```

6. Implementation Checklist

Phase 1: Core Forensics (Week 1-2)

- □ Set up evidence storage system
- □ Implement chain of custody

- □ Deploy disk forensics tools
- □ Create evidence acquisition procedures

Phase 2: Analysis Tools (Week 3-4)

- □ Implement memory forensics
- □ Deploy network forensics
- □ Set up malware analysis sandbox
- □ Create timeline analysis tools

Phase 3: Automation (Week 5-6)

- □ Automate evidence collection
- □ Implement analysis workflows
- □ Deploy ML models
- □ Create reporting automation

Phase 4: Integration (Week 7-8)

- □ Integrate with SIEM
- □ Connect to threat intelligence
- □ Set up case management
- □ Deploy visualization tools

7. Best Practices

Evidence Handling

1. **Write Protection**: Always use write blockers
2. **Hashing**: Calculate hashes before and after
3. **Documentation**: Document every action
4. **Copies**: Work on copies, preserve originals
5. **Encryption**: Encrypt evidence at rest

Analysis Standards

1. **Repeatability**: Ensure analysis can be repeated
2. **Tool Validation**: Validate all tools used
3. **Peer Review**: Have findings peer reviewed
4. **Documentation**: Document methodology
5. **Court Readiness**: Prepare for legal scrutiny

Reporting Guidelines

1. **Clarity**: Use clear, non-technical language
2. **Accuracy**: Verify all findings
3. **Completeness**: Include all relevant information
4. **Objectivity**: Remain unbiased
5. **Visualization**: Use charts and graphs effectively