06 Integration Guides

MWRASP Quantum Defense System

Generated: 2025-08-24 18:15:14

SECRET - AUTHORIZED PERSONNEL ONLY

MWRASP INTEGRATION GUIDES

Complete Technical Integration Documentation

QUICK START INTEGRATION

15-Minute Setup

```
 # 1. Download MWRASP
wget https://mwrasp.defense/download/mwrasp-latest.tar.gz
tar -xzf mwrasp-latest.tar.gz

# 2. Run installer
cd mwrasp
sudo ./install.sh --mode=overlay --quick-start

# 3. Verify installation
mwrasp status
# Expected: MWRASP Active - 127 agents online
```

```
# 4. Test quantum detection
mwrasp test quantum-attack --simulate
# Expected: Attack detected in <1ms, fragments expired
```

Docker Deployment

```
# docker-compose.yml
version: '3.8'
services:
  mwrasp-core:
    image: mwrasp/quantum-defense:latest
    ports:
      - "8443:8443"  # API
      - "9443:9443"  # WebSocket
      - "7443:7443"  # Agent communication
    volumes:
      - ./config:/etc/mwrasp
      - ./data:/var/lib/mwrasp
    environment:
      - MWRASP_MODE=production
      - FRAGMENT_EXPIRY=100ms
      - AGENT_COUNT=127
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: '4'
          memory: 8G
```

ENTERPRISE INTEGRATIONS

1. SIEM Integration

Splunk Integration

```
# splunk_connector.py
from mwrasp import MWRASPClient
import splunklib.client as client
```

```
class MWRASPSplunkConnector:
    def __init__(self):
        self.mwrasp = MWRASPClient(
            host='localhost',
            port=8443,
            api_key=os.environ['MWRASP_API_KEY']
        )
        self.splunk = client.connect(
            host='splunk.company.com',
            port=8089,
            username='admin',
            password=os.environ['SPLUNK_PASS']
        )

    def stream_quantum_events(self):
        """Stream quantum detection events to Splunk"""
        for event in self.mwrasp.stream_events():
            if event.type == 'quantum_detection':
                self.splunk.indexes['security'].submit(
                    event.to_json(),
                    sourcetype='mwrasp:quantum',
                    host=event.agent_id
                )
```

IBM QRadar Integration

```
// qradar_integration.js
const MWRASP = require('mwrasp-sdk');
const QRadar = require('qradar-api');

class MWRASPQRadarBridge {
    constructor() {
        this.mwrasp = new MWRASP.Client({
            endpoint: 'https://mwrasp.local:8443',
            apiKey: process.env.MWRASP_KEY
        });

        this.qradar = new QRadar({
            host: 'qradar.company.com',
            token: process.env.QRADAR_TOKEN
        });
    }

    async syncEvents() {
        // Subscribe to MWRASP events
        this.mwrasp.on('threat_detected', async (threat) => {
            await this.qradar.createOffense({
                description: `Quantum Attack: ${threat.type}`,
```

```
                severity: threat.quantum_probability * 10,
                categories: ['Quantum', 'APT', 'Nation-State'],
                credibility: 3,
                relevance: 5
            });
        });
    }
}
```

Elastic Security Integration

```
 # elastic_pipeline.yml
input:
  - module: mwrasp
    enabled: true
    var.hosts: ["mwrasp.local:9443"]
    var.ssl.certificate_authorities: ["/etc/mwrasp/ca.crt"]
    var.api_key: "${MWRASP_API_KEY}"

processors:
  - add_fields:
      target: ''
      fields:
        event.module: mwrasp
        event.dataset: quantum.defense

  - script:
      lang: javascript
      source: |
        function process(event) {
          if (event.Get("mwrasp.quantum_detected")) {
            event.Put("threat.indicator.type", "quantum");
            event.Put("threat.indicator.confidence", "high");
          }
        }

output.elasticsearch:
  hosts: ["elastic.local:9200"]
  index: "mwrasp-%{+yyyy.MM.dd}"
```

---

2. Cloud Platform Integration

AWS Integration

```
 # aws_integration.py
import boto3
from mwrasp import MWRASPCloud

class MWRASPAWSIntegration:
    def __init__(self):
        self.mwrasp = MWRASPCloud()
        self.guardduty = boto3.client('guardduty')
        self.security_hub = boto3.client('securityhub')

    def protect_s3_bucket(self, bucket_name):
        """Fragment S3 objects for quantum protection"""
        s3 = boto3.client('s3')

        # Enable MWRASP protection
        response = self.mwrasp.protect_storage(
            provider='aws',
            service='s3',
            resource=bucket_name,
            settings={
                'fragment_size': '1MB',
                'expiry_ms': 100,
                'jurisdictions': ['us-east-1', 'eu-west-1', 'ap-
southeast-1'],
                'quantum_detection': True
            }
        )

        # Configure S3 event notifications
        s3.put_bucket_notification_configuration(
            Bucket=bucket_name,
            NotificationConfiguration={
                'LambdaFunctionConfigurations': [{
                    'LambdaFunctionArn': response['lambda_arn'],
                    'Events': ['s3:ObjectCreated:*',
's3:ObjectRemoved:*']
                }]
            }
        )

    def integrate_with_guardduty(self):
        """Send MWRASP findings to GuardDuty"""
        findings = []
        for threat in self.mwrasp.get_threats():
            findings.append({
                'SchemaVersion': '2.0',
                'Id': threat.id,
                'ProductArn': 'arn:aws:securityhub:us-east-
1:123456789012:product/mwrasp/quantum-defense',
                'GeneratorId': 'mwrasp-quantum-detector',
                'AwsAccountId': '123456789012',
```

```
                'Types': ['Quantum Attack/Data Exfiltration'],
                'CreatedAt': threat.detected_at,
                'Severity': {
                    'Label': 'CRITICAL' if threat.quantum_probability
> 0.8 else 'HIGH'
                },
                'Title': f'Quantum Attack Detected:
{threat.attack_type}',
                'Description': threat.description
            })

        self.security_hub.batch_import_findings(Findings=findings)
```

Azure Integration

```
 // AzureIntegration.cs
using MWRASP.SDK;
using Azure.Security.KeyVault.Secrets;
using Azure.Identity;
using Microsoft.Azure.EventHubs;

public class MWRASPAzureIntegration
{
    private MWRASPClient mwrasp;
    private SecretClient keyVault;
    private EventHubClient eventHub;

    public MWRASPAzureIntegration()
    {
        mwrasp = new MWRASPClient(
            endpoint: "https://mwrasp.local:8443",
            apiKey: Environment.GetEnvironmentVariable("MWRASP_KEY")
        );

        keyVault = new SecretClient(
            new Uri("https://myvault.vault.azure.net/"),
            new DefaultAzureCredential()
        );

        eventHub = EventHubClient.CreateFromConnectionString(
            Environment.GetEnvironmentVariable("EVENTHUB_CONNECTION")
        );
    }

    public async Task ProtectKeyVault()
    {
        // Fragment all secrets for quantum protection
        await foreach (var secret in
keyVault.GetPropertiesOfSecretsAsync())
```

```
        {
            var value = await keyVault.GetSecretAsync(secret.Name);
            var protected = await mwrasp.FragmentDataAsync(
                data: value.Value.Value,
                expiry: TimeSpan.FromMilliseconds(100),
                jurisdictions: new[] { "Azure-US", "Azure-EU", "Azure-
Asia" }
            );

            // Store fragment map
            await keyVault.SetSecretAsync(
                $"{secret.Name}-mwrasp-map",
                protected.FragmentMap
            );
        }
    }

    public async Task StreamToSentinel()
    {
        // Stream MWRASP events to Azure Sentinel
        mwrasp.OnQuantumDetection += async (sender, e) =>
        {
            var eventData = new EventData(Encoding.UTF8.GetBytes(
                JsonSerializer.Serialize(new
                {
                    TimeGenerated = e.Timestamp,
                    Computer = e.AgentId,
                    EventID = 9001,
                    EventType = "QuantumAttackDetected",
                    Severity = "Critical",
                    AttackVector = e.AttackType,
                    QuantumProbability = e.Probability,
                    ResponseAction = e.ResponseTaken
                })
            ));

            await eventHub.SendAsync(eventData);
        };
    }
}
```

Google Cloud Integration

```
 // gcp_integration.go
package main

import (
    "cloud.google.com/go/pubsub"
    "cloud.google.com/go/storage"
```

```
    "github.com/mwrasp/sdk-go"
)

type MWRASPGCPIntegration struct {
    mwrasp   *mwrasp.Client
    storage  *storage.Client
    pubsub   *pubsub.Client
}

func (m *MWRASPGCPIntegration) ProtectCloudStorage(bucketName string)
error {
    bucket := m.storage.Bucket(bucketName)

    // Configure MWRASP protection
    protection := mwrasp.StorageProtection{
        Provider:     "gcp",
        Resource:     bucketName,
        FragmentSize: 1024 * 1024, // 1MB
        ExpiryMS:     100,
        Jurisdictions: []string{
            "us-central1",
            "europe-west1",
            "asia-northeast1",
        },
    }

    // Apply temporal fragmentation to all objects
    it := bucket.Objects(ctx, nil)
    for {
        attrs, err := it.Next()
        if err == iterator.Done {
            break
        }

        obj := bucket.Object(attrs.Name)
        reader,   := obj.NewReader(ctx)
        data, _ := ioutil.ReadAll(reader)

        fragments, _ := m.mwrasp.Fragment(data, protection)

        // Store fragments across regions
        for i, frag := range fragments {
            fragBucket := m.storage.Bucket(
                fmt.Sprintf("%s-mwrasp-%s", bucketName,
frag.Jurisdiction)
            )
            fragObj := fragBucket.Object(fmt.Sprintf("%s-frag-%d",
attrs.Name, i))
            writer := fragObj.NewWriter(ctx)
            writer.Write(frag.Data)
            writer.Close()
        }
```

```
    }

    return nil
}

func (m *MWRASPGCPIntegration) PublishToSecurityCommand() {
    topic := m.pubsub.Topic("security-events")

    // Subscribe to MWRASP events
    m.mwrasp.Subscribe(func(event mwrasp.Event) {
        if event.Type == mwrasp.QuantumDetection {
            msg := &pubsub.Message{
                Data: []byte(event.JSON()),
                Attributes: map[string]string{
                    "severity": "CRITICAL",
                    "type":     "quantum_attack",
                    "source":   "mwrasp",
                },
            }
            topic.Publish(ctx, msg)
        }
    })
}
```

---

3. Identity Provider Integration

Active Directory Integration

```
 # ActiveDirectory_Integration.ps1
Import-Module ActiveDirectory
Import-Module MWRASP

function Enable-MWRASPAuthentication {
    param(
        [string]$DomainController = "DC01.company.local",
        [string]$MWRASPServer = "mwrasp.company.local"
    )

    # Connect to MWRASP
    $mwrasp = Connect-MWRASP -Server $MWRASPServer -Credential (Get-
Credential)

    # Configure behavioral authentication for all users
    $users = Get-ADUser -Filter * -Properties *

    foreach ($user in $users) {
        # Create behavioral profile
```

```
        $profile = New-MWRASPProfile -Type "Behavioral" -Data @{
            UserPrincipalName = $user.UserPrincipalName
            SamAccountName = $user.SamAccountName
            Department = $user.Department
            Location = $user.Office
            LoginPatterns = Get-ADUserLoginHistory
$user.SamAccountName
        }

        # Enable temporal authentication
        Set-MWRASPAuthentication -Profile $profile -Settings @{
            FragmentExpiry = 100  # ms
            BehavioralFactors = @(
                "TypingCadence",
                "MouseMovement",
                "ApplicationUsage",
                "NetworkPatterns"
            )
            QuantumDetection = $true
            GeographicValidation = $true
        }

        # Update AD attributes
        Set-ADUser $user -Add @{
            "extensionAttribute1" = $profile.Id
            "extensionAttribute2" = "MWRASP-Protected"
        }
    }

    # Configure Group Policy for MWRASP
    New-GPO -Name "MWRASP Quantum Defense" | Set-GPRegistryValue -Key
"HKLM\Software\MWRASP" -ValueName "Enabled" -Value 1 -Type DWord
}

function Test-QuantumAuthentication {
    param([string]$Username)

    $result = Invoke-MWRASPTest -Type "QuantumAuth" -Target $Username
-Simulate

    if ($result.QuantumDetected) {
        # Immediate response
        Disable-ADAccount -Identity $Username
        Send-MailMessage -To "security@company.com" -Subject "QUANTUM
ATTACK DETECTED" -Body $result.Details

        # Fragment user data
        $userData = Get-ADUser $Username -Properties *
        $fragments = New-MWRASPFragment -Data ($userData | ConvertTo-
Json) -Expiry 100ms

        Write-EventLog -LogName "Security" -Source "MWRASP" -EventId
```

```
9001 -EntryType Error -Message "Quantum authentication attack blocked
for $Username"
    }

    return $result
}
```

Okta Integration

```
 // okta_integration.js
const OktaAuth = require('@okta/okta-auth-js');
const MWRASP = require('mwrasp-sdk');

class MWRASPOktaIntegration {
    constructor(config) {
        this.okta = new OktaAuth({
            issuer: config.oktaIssuer,
            clientId: config.oktaClientId,
            redirectUri: config.redirectUri
        });

        this.mwrasp = new MWRASP.Client({
            endpoint: config.mwraspEndpoint,
            apiKey: config.mwraspApiKey
        });

        this.setupHooks();
    }

    setupHooks() {
        // Pre-authentication hook
        this.okta.on('beforeAuthenticate', async (context) => {
            // Create behavioral snapshot
            const behavior = await this.mwrasp.captureBehavior({
                userId: context.username,
                sessionId: context.sessionId,
                factors: {
                    typing: context.typingPattern,
                    mouse: context.mouseMovements,
                    timing: context.interactionTiming,
                    device: context.deviceFingerprint
                }
            });

            // Check for quantum anomalies
            const quantumCheck = await this.mwrasp.detectQuantumAuth({
                behavior: behavior,
                threshold: 0.95
            });
```

```
            if (quantumCheck.detected) {
                // Quantum attack detected - fragment session
                await this.mwrasp.fragmentSession({
                    sessionId: context.sessionId,
                    expiry: 100, // ms
                    reason: 'Quantum authentication attempt detected'
                });

                throw new Error('Authentication blocked: Quantum
anomaly detected');
            }
        });

        // Post-authentication validation
        this.okta.on('afterAuthenticate', async (context) => {
            // Enable continuous behavioral monitoring
            const monitor = await this.mwrasp.startMonitoring({
                userId: context.user.id,
                sessionId: context.sessionToken,
                interval: 1000, // Check every second
                factors: [
                    'keystroke_dynamics',
                    'mouse_behavior',
                    'application_interaction',
                    'network_patterns'
                ]
            });

            monitor.on('anomaly', async (event) => {
                if (event.quantum_probability > 0.8) {
                    // Immediate response to quantum threat
                    await
this.okta.revokeSession(context.sessionToken);
                    await this.mwrasp.quarantineUser(context.user.id);
                }
            });
        });
    }

    async protectOktaUser(userId) {
        // Fragment user profile data
        const user = await this.okta.getUser(userId);

        const protection = await this.mwrasp.protectIdentity({
            provider: 'okta',
            userId: userId,
            data: user,
            settings: {
                fragmentation: {
                    size: 1024,
                    expiry: 100,
```

```
                    jurisdictions: ['US', 'EU', 'APAC']
                },
                behavioral: {
                    factors: 15,
                    threshold: 0.95,
                    learning: true
                },
                quantum: {
                    detection: true,
                    response: 'immediate',
                    canaryTokens: 10
                }
            }
        });

        return protection;
    }
}
```

---

4. Network Infrastructure Integration

Cisco Integration

```
 # cisco_integration.py
from netmiko import ConnectHandler
from mwrasp import NetworkDefense
import yaml

class MWRASPCiscoIntegration:
    def __init__(self, config_file):
        with open(config_file) as f:
            self.config = yaml.safe_load(f)

        self.mwrasp = NetworkDefense(
            api_endpoint=self.config['mwrasp']['endpoint'],
            api_key=self.config['mwrasp']['api_key']
        )

    def protect_catalyst_9000(self, switch_ip):
        """Integrate MWRASP with Catalyst 9000 series"""

        device = {
            'device_type': 'cisco_ios',
            'host': switch_ip,
            'username': self.config['cisco']['username'],
            'password': self.config['cisco']['password'],
            'secret': self.config['cisco']['enable_secret']
```

```
        }

        with ConnectHandler(**device) as net_connect:
            net_connect.enable()

            # Configure SPAN for MWRASP monitoring
            commands = [
                'monitor session 1 source interface range Gi1/0/1 -
48',
                'monitor session 1 destination interface Gi1/0/49',
                f'monitor session 1 filter ip access-group mwrasp-
quantum',

                # Create ACL for quantum detection
                'ip access-list extended mwrasp-quantum',
                'permit ip any any log',
                'exit',

                # Configure NetFlow for behavioral analysis
                'flow record mwrasp-record',
                'match ipv4 source address',
                'match ipv4 destination address',
                'match transport source-port',
                'match transport destination-port',
                'collect counter bytes',
                'collect counter packets',
                'collect timestamp absolute first',
                'collect timestamp absolute last',
                'exit',

                'flow exporter mwrasp-exporter',
                f'destination {self.config["mwrasp"]
["collector_ip"]}',
                'source Loopback0',
                'transport udp 2055',
                'exit',

                'flow monitor mwrasp-monitor',
                'record mwrasp-record',
                'exporter mwrasp-exporter',
                'cache timeout active 1',
                'exit',

                # Apply to interfaces
                'interface range Gi1/0/1 - 48',
                'ip flow monitor mwrasp-monitor input',
                'ip flow monitor mwrasp-monitor output',
                'exit'
            ]

            output = net_connect.send_config_set(commands)
```

```
            # Register switch with MWRASP
            self.mwrasp.register_device({
                'type': 'cisco_catalyst_9000',
                'ip': switch_ip,
                'location': self.config['cisco']['location'],
                'capabilities': ['netflow', 'span', 'acl'],
                'quantum_monitoring': True
            })

        return output

    def configure_asa_quantum_defense(self, firewall_ip):
        """Configure Cisco ASA for quantum threat defense"""

        device = {
            'device_type': 'cisco_asa',
            'host': firewall_ip,
            'username': self.config['asa']['username'],
            'password': self.config['asa']['password'],
            'secret': self.config['asa']['enable_secret']
        }

        with ConnectHandler(**device) as net_connect:
            net_connect.enable()

            commands = [
                # Configure threat detection
                'threat-detection basic-threat',
                'threat-detection statistics',
                'threat-detection statistics tcp-intercept rate-
interval 30 burst-rate 400 average-rate 200',

                # Configure MWRASP integration
                f'logging host inside {self.config["mwrasp"]
["syslog_ip"]} 17/1514',
                'logging trap debugging',
                'logging device-id hostname',

                # Create quantum detection class-map
                'class-map mwrasp-quantum-class',
                'match any',
                'exit',

                # Create policy for quantum threats
                'policy-map mwrasp-quantum-policy',
                'class mwrasp-quantum-class',
                'set connection timeout embryonic 0:0:30',
                'set connection timeout half-closed 0:10:0',
                'set connection timeout tcp 1:0:0',
                'set connection per-client-max 100',
                'exit',
```

```
                # Apply to global policy
                'policy-map global_policy',
                'class mwrasp-quantum-class',
                'inspect mwrasp-quantum',
                'exit'
            ]

            output = net_connect.send_config_set(commands)

        return output
```

Palo Alto Networks Integration

```
 # palo_alto_integration.py
from pandevice import firewall, policies, objects
from mwrasp import ThreatIntelligence

class MWRASPPaloAltoIntegration:
    def __init__(self, pa_host, pa_key, mwrasp_host):
        self.fw = firewall.Firewall(pa_host, api_key=pa_key)
        self.mwrasp = ThreatIntelligence(mwrasp_host)

    def create_quantum_defense_profile(self):
        """Create custom threat profile for quantum attacks"""

        # Create custom URL categories for quantum C2
        quantum_urls = objects.CustomUrlCategory(
            name='mwrasp-quantum-c2',
            url_value=[
                '*.quantum-malware.test',
                '*.qbit-exfil.test'
            ]
        )
        self.fw.add(quantum_urls)

        # Create security profile for quantum threats
        quantum_profile = policies.SecurityProfileGroup(
            name='MWRASP-Quantum-Defense',
            virus='strict',
            spyware='strict',
            vulnerability='strict',
            url_filtering='mwrasp-quantum-filter',
            file_blocking='strict',
            wildfire_analysis='public-cloud',
            data_filtering='mwrasp-quantum-dlp'
        )
        self.fw.add(quantum_profile)

        # Create security rule
```

```
        quantum_rule = policies.SecurityRule(
            name='Block-Quantum-Attacks',
            fromzone=['any'],
            tozone=['any'],
            source=['any'],
            destination=['any'],
            application=['any'],
            service=['any'],
            action='deny',
            profile_setting=quantum_profile,
            log_setting='mwrasp-quantum-logs'
        )
        self.fw.add(quantum_rule)

        # Commit configuration
        self.fw.commit()

    def sync_quantum_intelligence(self):
        """Sync quantum threat intelligence with Palo Alto"""

        # Get latest quantum indicators from MWRASP
        indicators = self.mwrasp.get_quantum_indicators()

        # Create external dynamic list
        edl = objects.ExternalDynamicList(
            name='mwrasp-quantum-ioc',
            type='ip',
            recurring='five-minute',
            url=f'{self.mwrasp.host}/api/v1/quantum-ioc/paloalto',
            description='MWRASP Quantum Attack Indicators'
        )
        self.fw.add(edl)

        # Create corresponding security policy
        for ioc in indicators:
            if ioc['type'] == 'quantum_signature':
                # Add custom App-ID for quantum attacks
                app = objects.ApplicationObject(
                    name=f'quantum-attack-{ioc["id"]}',
                    category='malware',
                    subcategory='quantum-malware',
                    technology='quantum-computing',
                    risk=5,
                    default_port='any'
                )
                self.fw.add(app)

        self.fw.commit()
```

---

5. Database Integration

PostgreSQL Integration

```
 -- postgresql_integration.sql
-- MWRASP PostgreSQL Extension for Quantum Defense

CREATE EXTENSION IF NOT EXISTS mwrasp_quantum_defense;

-- Create schema for MWRASP
CREATE SCHEMA IF NOT EXISTS mwrasp;

-- Temporal fragmentation function
CREATE OR REPLACE FUNCTION mwrasp.fragment_data(
    p_data BYTEA,
    p_expiry_ms INTEGER DEFAULT 100,
    p_jurisdictions TEXT[] DEFAULT ARRAY['US', 'EU', 'ASIA']
) RETURNS TABLE (
    fragment_id UUID,
    fragment_data BYTEA,
    jurisdiction TEXT,
    expiry_time TIMESTAMP
) AS $$
BEGIN
    -- Fragment data across jurisdictions
    RETURN QUERY
    WITH fragments AS (
        SELECT
            gen_random_uuid() as frag_id,
            substring(p_data FROM (i-1)*1024 + 1 FOR 1024) as
frag_data,
            p_jurisdictions[1 + (i % array_length(p_jurisdictions,
1))] as jurisdiction,
            NOW() + (p_expiry_ms || ' milliseconds')::INTERVAL as
expiry
        FROM generate_series(1, octet_length(p_data) / 1024 + 1) i
    )
    SELECT * FROM fragments;
END;
$$ LANGUAGE plpgsql;

-- Behavioral authentication tracking
CREATE TABLE mwrasp.behavioral_profiles (
    user_id UUID PRIMARY KEY,
    typing_pattern JSONB,
    query_patterns JSONB,
    access_times JSONB,
    quantum_risk_score FLOAT DEFAULT 0.0,
    last_updated TIMESTAMP DEFAULT NOW()
```

```
);

-- Quantum detection triggers
CREATE OR REPLACE FUNCTION mwrasp.detect_quantum_query()
RETURNS EVENT_TRIGGER AS $$
DECLARE
    query_complexity INTEGER;
    quantum_probability FLOAT;
BEGIN
    -- Analyze query for quantum patterns
    SELECT mwrasp.calculate_query_complexity(current_query()) INTO
query_complexity;

    IF query_complexity > 1000000 THEN
        -- Possible quantum attack - check patterns
        SELECT mwrasp.quantum_pattern_match(current_query()) INTO
quantum_probability;

        IF quantum_probability > 0.8 THEN
            -- Fragment all sensitive data immediately
            PERFORM mwrasp.emergency_fragment_all();

            -- Log quantum detection
            INSERT INTO mwrasp.quantum_detections (
                detected_at,
                query_text,
                probability,
                response_action
            ) VALUES (
                NOW(),
                current_query(),
                quantum_probability,
                'DATA_FRAGMENTED'
            );

            -- Terminate suspicious connection
            PERFORM pg_terminate_backend(pg_backend_pid());
        END IF;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER mwrasp_quantum_detector
    ON ddl_command_start
    EXECUTE FUNCTION mwrasp.detect_quantum_query();

-- Row-level security for quantum defense
ALTER TABLE sensitive_data ENABLE ROW LEVEL SECURITY;

CREATE POLICY mwrasp_quantum_policy ON sensitive_data
    FOR ALL
    USING (
```

```
        mwrasp.check_quantum_state() = 'SAFE'
        AND
mwrasp.validate_jurisdiction(current_setting('mwrasp.user_location'))
    );
```

MongoDB Integration

```
 // mongodb_integration.js
const { MongoClient } = require('mongodb');
const MWRASP = require('mwrasp-sdk');

class MWRASPMongoDBIntegration {
    constructor(mongoUri, mwraspConfig) {
        this.client = new MongoClient(mongoUri);
        this.mwrasp = new MWRASP.Client(mwraspConfig);
        this.setupMiddleware();
    }

    setupMiddleware() {
        // Intercept all database operations
        const originalFind = this.client.db().collection().find;

        this.client.db().collection().find = async function(...args) {
            // Check for quantum attack patterns
            const quantumCheck = await this.mwrasp.analyzeQuery({
                operation: 'find',
                args: args,
                timestamp: Date.now()
            });

            if (quantumCheck.quantum_detected) {
                // Fragment the collection
                await this.fragmentCollection(this.collectionName);
                throw new Error('Quantum attack detected - data
fragmented');
            }

            return originalFind.apply(this, args);
        };
    }

    async fragmentCollection(collectionName) {
        const db = this.client.db();
        const collection = db.collection(collectionName);

        // Read all documents
        const documents = await collection.find({}).toArray();

        // Fragment each document
```

```
        for (const doc of documents) {
            const fragments = await this.mwrasp.fragment({
                data: JSON.stringify(doc),
                expiry: 100, // ms
                jurisdictions: ['mongodb-us', 'mongodb-eu', 'mongodb-
asia']
            });

            // Store fragments in separate collections
            for (const fragment of fragments) {
                await
db.collection(`${collectionName}_frag_${fragment.jurisdiction}`)
                    .insertOne({
                        original_id: doc._id,
                        fragment_id: fragment.id,
                        data: fragment.data,
                        expiry: fragment.expiry,
                        jurisdiction: fragment.jurisdiction
                    });
            }
        }

        // Delete original collection
        await collection.drop();

        // Create view that reconstructs data only for authorized
access
        await db.createCollection(`${collectionName}`, {
            viewOn: `${collectionName}_fragments`,
            pipeline: [
                { $match: { expiry: { $gt: new Date() } } },
                { $group: { _id: '$original_id', fragments: { $push:
'$$ROOT' } } },
                { $lookup: {
                    from: 'mwrasp_auth',
                    pipeline: [{ $match: { quantum_safe: true } }],
                    as: 'auth'
                }},
                { $match: { auth: { $ne: [] } } }
            ]
        });
    }

    async enableQuantumDefense(dbName) {
        const db = this.client.db(dbName);

        // Add quantum detection to change streams
        const changeStream = db.watch([], {
            fullDocument: 'updateLookup'
        });

        changeStream.on('change', async (change) => {
```

```
            const analysis = await this.mwrasp.analyzeChange(change);

            if (analysis.quantum_probability > 0.7) {
                // Quantum activity detected
                console.error(`QUANTUM ALERT: ${analysis.details}`);

                // Fragment affected data
                if (change.fullDocument) {
                    await this.fragmentDocument(
                        change.ns.coll,
                        change.fullDocument._id
                    );
                }

                // Block the operation source
                await this.mwrasp.blockSource(change.operationSource);
            }
        });
    }
}
```

---

6. Container & Kubernetes Integration

Kubernetes Integration

```
 # kubernetes integration.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: mwrasp-system

---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: mwrasp-agent
  namespace: mwrasp-system
spec:
  selector:
    matchLabels:
      app: mwrasp-agent
  template:
    metadata:
      labels:
        app: mwrasp-agent
    spec:
      hostNetwork: true
```

```
      hostPID: true
      hostIPC: true
      containers:
      - name: mwrasp-agent
        image: mwrasp/agent:latest
        securityContext:
          privileged: true
        env:
        - name: MWRASP_MODE
          value: "kubernetes"
        - name: FRAGMENT_EXPIRY
          value: "100ms"
        - name: AGENT_COUNT
          value: "127"
        - name: QUANTUM_DETECTION
          value: "enabled"
        volumeMounts:
        - name: docker-sock
          mountPath: /var/run/docker.sock
        - name: proc
          mountPath: /host/proc
        - name: sys
          mountPath: /host/sys
      volumes:
      - name: docker-sock
        hostPath:
          path: /var/run/docker.sock
      - name: proc
        hostPath:
          path: /proc
      - name: sys
        hostPath:
          path: /sys

---
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: mwrasp-quantum-validator
webhooks:
- name: quantum.mwrasp.io
  clientConfig:
    service:
      name: mwrasp-webhook
      namespace: mwrasp-system
      path: "/validate"
    caBundle: LS0tLS1CRUdJTi... # Base64 encoded CA
  rules:
  - operations: ["CREATE", "UPDATE"]
    apiGroups: ["*"]
    apiVersions: ["*"]
    resources: ["pods", "deployments", "services"]
```

```
  admissionReviewVersions: ["v1"]
  sideEffects: None
  failurePolicy: Fail

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: mwrasp-config
  namespace: mwrasp-system
data:
  mwrasp.yaml: |
    defense:
      temporal_fragmentation:
        enabled: true
        expiry_ms: 100
        fragment_size: 1024
      behavioral_auth:
        enabled: true
        factors: 15
      quantum_detection:
        enabled: true
        canary_tokens: 10
        response_time_ms: 1
      jurisdictions:
        - us-east-1
        - eu-west-1
        - ap-southeast-1
    kubernetes:
      cluster_name: production
      monitor_namespaces:
        - default
        - production
        - staging
      exclude_namespaces:
        - kube-system
        - kube-public
```

Docker Integration

```
 # Dockerfile.mwrasp
FROM alpine:latest

# Install MWRASP runtime
RUN apk add --no-cache \
    python3 \
    py3-pip \
    openssl \
    libffi-dev
```

```
# Install MWRASP
COPY mwrasp-runtime /usr/local/bin/mwrasp
RUN chmod +x /usr/local/bin/mwrasp

# Configure MWRASP as entrypoint wrapper
ENTRYPOINT ["/usr/local/bin/mwrasp", "protect"]
CMD ["--fragment-expiry", "100ms", "--quantum-detection", "on"]

# Example usage in docker-compose.yml:
# services:
#   app:
#     build:
#       context: .
#       dockerfile: Dockerfile.mwrasp
#     environment:
#       MWRASP_API_KEY: ${MWRASP_KEY}
#       MWRASP_MODE: container
#     volumes:
#       - /var/run/docker.sock:/var/run/docker.sock:ro
```

API INTEGRATION EXAMPLES

REST API Integration

```
 # rest api integration.py
import requests
from typing import Dict, Any
import hmac
import hashlib
import time

class MWRASPAPIClient:
    def __init__(self, base_url: str, api_key: str, api_secret: str):
        self.base_url = base_url
        self.api_key = api_key
        self.api_secret = api_secret
        self.session = requests.Session()

    def _sign_request(self, method: str, path: str, body: str = "") ->
Dict[str, str]:
        timestamp = str(int(time.time()))
        message = f"{method}{path}{timestamp}{body}"
        signature = hmac.new(
            self.api_secret.encode(),
```

```
            message.encode(),
            hashlib.sha256
        ).hexdigest()

        return {
            'X-MWRASP-Key': self.api_key,
            'X-MWRASP-Signature': signature,
            'X-MWRASP-Timestamp': timestamp
        }

    def fragment_data(self, data: bytes, expiry_ms: int = 100) ->
Dict[str, Any]:
        """Fragment data for quantum protection"""
        path = '/api/v1/fragment'
        headers = self._sign_request('POST', path, data.decode('utf-
8'))

        response = self.session.post(
            f"{self.base_url}{path}",
            headers=headers,
            json={
                'data': data.hex(),
                'expiry_ms': expiry_ms,
                'jurisdictions': ['US', 'EU', 'ASIA']
            }
        )
        return response.json()

    def detect_quantum_attack(self, sample: bytes) -> Dict[str, Any]:
        """Check if data shows quantum attack signatures"""
        path = '/api/v1/quantum/detect'
        headers = self._sign_request('POST', path)

        response = self.session.post(
            f"{self.base_url}{path}",
            headers=headers,
            json={'sample': sample.hex()}
        )
        return response.json()
```

WebSocket Integration

```
 // websocket_integration.js
const WebSocket = require('ws');
const crypto = require('crypto');

class MWRASPWebSocketClient {
    constructor(config) {
```

```
        this.config = config;
        this.ws = null;
        this.handlers = new Map();
        this.reconnectAttempts = 0;
    }

    connect() {
        const url = 
`wss://${this.config.host}:${this.config.port}/ws`;
        this.ws = new WebSocket(url, {
            headers: {
                'X-MWRASP-Key': this.config.apiKey
            }
        });

        this.ws.on('open', () => {
            console.log('Connected to MWRASP');
            this.authenticate();
            this.reconnectAttempts = 0;
        });

        this.ws.on('message', (data) => {
            const message = JSON.parse(data);
            this.handleMessage(message);
        });

        this.ws.on('close', () => {
            this.reconnect();
        });

        this.ws.on('error', (error) => {
            console.error('MWRASP WebSocket error:', error);
        });
    }

    authenticate() {
        const timestamp = Date.now();
        const signature = crypto
            .createHmac('sha256', this.config.apiSecret)
            .update(`${this.config.apiKey}${timestamp}`)
            .digest('hex');

        this.send({
            type: 'auth',
            apiKey: this.config.apiKey,
            timestamp: timestamp,
            signature: signature
        });
    }

    handleMessage(message) {
        switch(message.type) {
```

```
            case 'quantum_detection':
                this.onQuantumDetection(message);
                break;
            case 'fragment_expired':
                this.onFragmentExpired(message);
                break;
            case 'agent_evolution':
                this.onAgentEvolution(message);
                break;
            case 'threat_response':
                this.onThreatResponse(message);
                break;
            default:
                if (this.handlers.has(message.type)) {
                    this.handlers.get(message.type)(message);
                }
        }
    }

    onQuantumDetection(data) {
        console.error(`QUANTUM ATTACK DETECTED: ${data.details}`);
        // Immediate response
        this.send({
            type: 'fragment_all',
            urgency: 'immediate',
            expiry: 50 // Even faster expiry under attack
        });
    }

    subscribe(event, handler) {
        this.handlers.set(event, handler);
        this.send({
            type: 'subscribe',
            event: event
        });
    }

    send(data) {
        if (this.ws && this.ws.readyState === WebSocket.OPEN) {
            this.ws.send(JSON.stringify(data));
        }
    }

    reconnect() {
        if (this.reconnectAttempts < 10) {
            setTimeout(() => {
                this.reconnectAttempts++;
                console.log(`Reconnecting... (attempt
${this.reconnectAttempts})`);
                this.connect();
            }, Math.min(1000 * Math.pow(2, this.reconnectAttempts),
30000));
```

```
        }
    }
}
```

TROUBLESHOOTING GUIDE

Common Integration Issues

Issue: Quantum Detection False Positives

```
# Solution: Tune detection sensitivity
mwrasp.configure_quantum_detection(
    sensitivity='balanced',  # Options: aggressive, balanced, conservative
    whitelist_patterns=[
        'legitimate_quantum_research',
        'quantum_simulator_testing'
    ],
    learning_mode=True  # Enable for first 30 days
)
```

Issue: Fragment Expiry Too Fast

```
# Solution: Implement adaptive expiry
def adaptive_fragment_expiry(threat_level):
    expiry_map = {
        'none': 1000,      # 1 second
        'low': 500,        # 500ms
        'medium': 200,     # 200ms
        'high': 100,       # 100ms
        'critical': 50     # 50ms
    }
    return expiry_map.get(threat_level, 100)
```

Issue: Agent Evolution Consuming Resources

```
# Solution: Limit agent evolution rate
mwrasp config set agent.evolution.max_rate 10
mwrasp config set agent.evolution.resource_limit 4GB
mwrasp config set agent.evolution.cpu_limit 50%
```

Issue: Jurisdiction Compliance Conflicts

```
# Solution: Configure jurisdiction rules
jurisdictions:
  priority:
    - Switzerland  # Privacy first
    - Iceland      # Data protection
    - Singapore    # Business friendly
  exclude:
    - China        # If data sovereignty required
    - Russia       # If sanctions apply
  compliance:
    GDPR: enabled
    CCPA: enabled
    HIPAA: enabled
```

PERFORMANCE OPTIMIZATION

Optimization Strategies

1. Fragment Size Optimization

```
def optimize_fragment_size(data_size, network_latency, threat_level):
    """Calculate optimal fragment size"""

    # Base calculation
    base_size = 1024  # 1KB default

    # Adjust for data size
    if data_size > 1_000_000:  # > 1MB
        base_size = 4096
    elif data_size > 10_000_000:  # > 10MB
        base_size = 8192
```

```
    # Adjust for network latency
    if network_latency > 50:  # High latency
        base_size = base_size // 2

    # Adjust for threat level
    if threat_level == 'critical':
        base_size = base_size // 4  # Smaller fragments = harder to
capture

    return base_size
```

2. Agent Distribution Strategy

```
def distribute_agents(topology, threat_model):
    """Optimize agent placement"""

    distribution = {
        'edge': int(127 * 0.4),      # 40% at edge
        'core': int(127 * 0.3),      # 30% at core
        'database': int(127 * 0.2),  # 20% at data layer
        'floating': int(127 * 0.1)   # 10% adaptive
    }

    if threat_model == 'insider':
        distribution['database'] += distribution['floating']
        distribution['floating'] = 0
    elif threat_model == 'external':
        distribution['edge'] += distribution['floating']
        distribution['floating'] = 0

    return distribution
```

VALIDATION & TESTING

Integration Test Suite

```
 # test_integration.py
import pytest
from mwrasp import MWRASPClient

class TestMWRASPIntegration:
```

```
    @pytest.fixture
    def client(self):
        return MWRASPClient('http://localhost:8443')

    def test_quantum_detection(self, client):
        """Test quantum attack detection"""
        # Simulate quantum pattern
        quantum_data = b'\x00\x01' * 1000  # Simplified quantum
signature

        result = client.detect_quantum(quantum_data)
        assert result['detected'] == True
        assert result['response_time'] < 1  # < 1ms

    def test_fragment_expiry(self, client):
        """Test 100ms fragment expiry"""
        data = b'sensitive_data'
        fragments = client.fragment(data, expiry=100)

        # Immediate retrieval should work
        retrieved = client.reconstruct(fragments['id'])
        assert retrieved == data

        # After 100ms, should fail
        time.sleep(0.101)
        with pytest.raises(FragmentExpiredException):
            client.reconstruct(fragments['id'])

    def test_agent_evolution(self, client):
        """Test agent evolution under attack"""
        initial_agents = client.get_agent_count()
        assert initial_agents == 127

        # Simulate attack
        client.simulate_attack('quantum_brute_force')
        time.sleep(1)

        # Agents should evolve
        evolved_agents = client.get_agent_count()
        assert evolved_agents > initial_agents

    def test_behavioral_auth(self, client):
        """Test behavioral authentication"""
        # Establish baseline
        baseline = client.create_behavioral_profile(
            user='test_user',
            samples=100
        )

        # Test normal behavior
        assert client.verify_behavior('test_user', normal_behavior) ==
```

```
True

        # Test anomalous behavior (potential quantum spoofing)
        assert client.verify_behavior('test_user', quantum_spoofed) ==
False
```

Integration Guide Version: 2.0 **Last Updated**: February 2024 **Total Integrations**: 50+ **Average Integration Time**: 2-4 hours **Support**: integration@mwrasp.defense

---

Document: 06_INTEGRATION_GUIDES.md | **Generated:** 2025-08-24 18:15:14