Technical Integration: Qualys, Tenable, Rapid7 to CPF Framework

API Integration Architecture

```
# config.yaml
qualys:
  api_url: "https://qualysapi.qualys.com/api/2.0/fo/"
  username: "${QUALYS_USER}"
  password: "${QUALYS_PASS}"

tenable:
  api_url: "https://cloud.tenable.com"
  access_key: "${TENABLE_ACCESS}"
  secret_key: "${TENABLE_SECRET}"

rapid7:
  api_url: "https://api.insight.rapid7.com/vm/v4/"
  api_key: "${RAPID7_KEY}"
```

Data Collection Pipeline

Step 1: Extract Raw Data from Each Scanner

```
import requests
from datetime import datetime, timedelta
import pandas as pd

class QualysExtractor:
    def __init__(self, config):
        self.session = requests.Session()
        self.session.auth = (config['username'], config['password'])
        self.base_url = config['api_url']

    def get_vulnerability_data(self, days_back=90):
        # Get host-level vulnerability data
        endpoint = f"{self.base_url}asset/host/vm/detection/"
        params = {
            'action': 'list',
            'show_results': 1,
            'output_format': 'json',
            'vm_scan_date_after': (datetime.now() -
timedelta(days=days_back)).isoformat(),
            'include_ignored': 1,
            'include_disabled': 1
```

```
        }

        response = self.session.get(endpoint, params=params)
        data = response.json()

        # Extract key fields for CPF analysis
        vulnerabilities = []
        for host in data['HOST_LIST']['HOST']:
            host_id = host['ID']
            for detection in host.get('DETECTION_LIST', {}).get('DETECTION', []):
                vulnerabilities.append({
                    'source': 'qualys',
                    'host_id': host_id,
                    'hostname': host.get('DNS', ''),
                    'ip': host.get('IP'),
                    'qid': detection['QID'],
                    'cve': detection.get('CVE_ID', ''),
                    'severity': detection.get('SEVERITY', 0),
                    'first_detected': detection.get('FIRST_FOUND_DATETIME'),
                    'last_detected': detection.get('LAST_FOUND_DATETIME'),
                    'times_detected': detection.get('TIMES_FOUND', 1),
                    'status': detection.get('STATUS'),
                    'ignored': detection.get('IS_IGNORED', False),
                    'disabled': detection.get('IS_DISABLED', False),
                    'patch_available': detection.get('PATCHABLE', False)
                })

        return pd.DataFrame(vulnerabilities)

class TenableExtractor:
    def __init__(self, config):
        self.headers = {
            'X-ApiKeys': f'accessKey={config["access_key"]};secretKey=
{config["secret_key"]}',
            'Content-Type': 'application/json'
        }
        self.base_url = config['api_url']

    def get_vulnerability_data(self, days_back=90):
        # Export vulnerabilities
        export_endpoint = f"{self.base_url}/vulns/export"

        # Create export request
        export_request = {
            'num_assets': 'all',
            'filters': {
                'since': int((datetime.now() - timedelta(days=days_back)).timestamp())
            }
        }
```

```
        response = requests.post(export_endpoint, json=export_request,
headers=self.headers)
        export_uuid = response.json()['export_uuid']

        # Poll for completion and download
        status_endpoint = f"{self.base_url}/vulns/export/{export_uuid}/status"
        while True:
            status = requests.get(status_endpoint, headers=self.headers).json()
            if status['status'] == 'FINISHED':
                break
            time.sleep(5)

        # Download chunks
        chunks_endpoint = f"{self.base_url}/vulns/export/{export_uuid}/chunks/1"
        data = requests.get(chunks_endpoint, headers=self.headers).json()

        vulnerabilities = []
        for vuln in data['vulnerabilities']:
            vulnerabilities.append({
                'source': 'tenable',
                'host_id': vuln['asset']['uuid'],
                'hostname': vuln['asset']['hostname'],
                'ip': vuln['asset']['ipv4'],
                'plugin_id': vuln['plugin']['id'],
                'cve': vuln.get('cve', [''])[0] if vuln.get('cve') else '',
                'severity': vuln['severity'],
                'first_detected': vuln['first_found'],
                'last_detected': vuln['last_found'],
                'state': vuln['state'],
                'has_patch': vuln['plugin'].get('has_patch', False),
                'exploit_available': vuln['plugin'].get('exploit_available', False),
                'exploited_by_malware': vuln['plugin'].get('exploited_by_malware',
False)
            })

        return pd.DataFrame(vulnerabilities)

class Rapid7Extractor:
    def __init__(self, config):
        self.headers = {
            'X-Api-Key': config['api_key'],
            'Content-Type': 'application/json'
        }
        self.base_url = config['api_url']

    def get_vulnerability_data(self, days_back=90):
        # Get vulnerability findings
        endpoint = f"{self.base_url}vulnerabilities"
```

```
        params = {
            'size': 1000,
            'sort': 'riskScore,DESC'
        }

        all_vulnerabilities = []
        page = 0

        while True:
            params['page'] = page
            response = requests.get(endpoint, params=params, headers=self.headers)
            data = response.json()

            for vuln in data['data']:
                for instance in vuln.get('instances', []):
                    all_vulnerabilities.append({
                        'source': 'rapid7',
                        'host_id': instance['assetId'],
                        'hostname': instance.get('hostname', ''),
                        'ip': instance.get('ip', ''),
                        'cve': vuln.get('cveId', ''),
                        'severity': vuln.get('severity', ''),
                        'risk_score': vuln.get('riskScore', 0),
                        'first_discovered': instance.get('discoveredDate'),
                        'status': instance.get('status'),
                        'proof': instance.get('proof', ''),
                        'exploitability': vuln.get('exploitability', ''),
                        'malware_kits': len(vuln.get('malwareKits', []))
                    })

            if not data.get('links', {}).get('next'):
                break
            page += 1

        return pd.DataFrame(all_vulnerabilities)
```

Step 2: Normalize and Merge Data

```
class DataNormalizer:
    def __init__(self):
        self.severity_mapping = {
            # Normalize severity across platforms
            'qualys': {5: 'Critical', 4: 'High', 3: 'Medium', 2: 'Low', 1: 'Info'},
            'tenable': {'critical': 'Critical', 'high': 'High', 'medium': 'Medium',
'low': 'Low'},
            'rapid7': {'Critical': 'Critical', 'Severe': 'High', 'Moderate': 'Medium',
'Low': 'Low'}
        }
```

```
    def merge_scanner_data(self, qualys_df, tenable_df, rapid7_df):
        # Normalize timestamps
        for df in [qualys_df, tenable_df, rapid7_df]:
            df['first_detected'] = pd.to_datetime(df['first_detected'])
            df['last_detected'] = pd.to_datetime(df['last_detected'])

        # Create unified host identifier
        def create_host_key(row):
            # Use IP as primary key, hostname as fallback
            return row['ip'] if pd.notna(row['ip']) else row['hostname']

        for df in [qualys_df, tenable_df, rapid7_df]:
            df['host_key'] = df.apply(create_host_key, axis=1)

        # Merge on CVE + host_key
        merged = pd.concat([qualys_df, tenable_df, rapid7_df], ignore_index=True)

        # Group by host_key and CVE to consolidate findings
        consolidated = merged.groupby(['host_key', 'cve']).agg({
            'first_detected': 'min',  # Earliest detection
            'last_detected': 'max',   # Most recent detection
            'severity': 'max',  # Highest severity
            'source': lambda x: list(x),  # All sources that found it
            'times_detected': 'sum',  # Total detections
            'patch_available': 'max',  # If any source says patch available
            'exploit_available': 'max',  # If any source says exploit available
            'ignored': 'min',  # If any source has it active (not ignored)
            'status': lambda x: 'ACTIVE' if 'ACTIVE' in x.values else 'RESOLVED'
        }).reset_index()

        return consolidated
```

Step 3: Pattern Detection Engine

```
class CPFPatternEngine:
    def __init__(self, consolidated_data):
        self.data = consolidated_data
        self.patterns = {}

    def detect_manic_defense(self):
        """
        Detect: Patches only after PoC/news, ignored before
        """
        # Get PoC publication dates from threat intel
        poc_dates = self.get_poc_dates()  # External threat intel API

        manic_score = 0
```

```
        evidence = []

        for cve in self.data['cve'].unique():
            cve_data = self.data[self.data['cve'] == cve]

            if cve in poc_dates:
                poc_date = poc_dates[cve]

                # Check if ignored before PoC, patched after
                before_poc = cve_data[cve_data['first_detected'] < poc_date]
                after_poc = cve_data[cve_data['first_detected'] >= poc_date]

                if len(before_poc) > 0 and len(after_poc) > 0:
                    avg_patch_time_before = (before_poc['last_detected'] -
before_poc['first_detected']).mean().days
                    avg_patch_time_after = (after_poc['last_detected'] -
after_poc['first_detected']).mean().days

                    if avg_patch_time_before > 30 and avg_patch_time_after < 2:
                        manic_score += 1
                        evidence.append({
                            'cve': cve,
                            'before_poc_days': avg_patch_time_before,
                            'after_poc_days': avg_patch_time_after
                        })

        return {
            'pattern': 'MANIC_DEFENSE',
            'score': min(manic_score * 0.1, 1.0),
            'evidence': evidence,
            'cpf_category': '[8.6]'
        }

    def detect_splitting(self):
        """
        Detect: Same CVE treated differently on different host types
        """
        # Classify hosts by type
        self.data['host_type'] = self.data['host_key'].apply(self.classify_host)

        splitting_score = 0
        evidence = []

        for cve in self.data['cve'].unique():
            cve_hosts = self.data[self.data['cve'] == cve]

            if len(cve_hosts['host_type'].unique()) > 1:
                # Calculate patch rate by host type
                patch_rates = {}
```

```
                for host_type in cve_hosts['host_type'].unique():
                    type_data = cve_hosts[cve_hosts['host_type'] == host_type]
                    patched = len(type_data[type_data['status'] == 'RESOLVED'])
                    total = len(type_data)
                    patch_rates[host_type] = patched / total if total > 0 else 0

                # Check for splitting pattern (>70% difference)
                if max(patch_rates.values()) - min(patch_rates.values()) > 0.7:
                    splitting_score += 1
                    evidence.append({
                        'cve': cve,
                        'patch_rates': patch_rates,
                        'good_object': max(patch_rates, key=patch_rates.get),
                        'bad_object': min(patch_rates, key=patch_rates.get)
                    })

        return {
            'pattern': 'SPLITTING',
            'score': min(splitting_score * 0.15, 1.0),
            'evidence': evidence,
            'cpf_category': '[4.9]'
        }

    def detect_repetition_compulsion(self):
        """
        Detect: CVE patched and returns multiple times
        """
        repetition_score = 0
        evidence = []

        for host in self.data['host_key'].unique():
            host_data = self.data[self.data['host_key'] == host]

            for cve in host_data['cve'].unique():
                cve_timeline = self.build_cve_timeline(host, cve)

                # Count patch->vulnerable cycles
                cycles = 0
                for i in range(len(cve_timeline) - 1):
                    if cve_timeline[i]['status'] == 'RESOLVED' and cve_timeline[i+1]
['status'] == 'ACTIVE':
                        cycles += 1

                if cycles >= 3:
                    repetition_score += cycles
                    evidence.append({
                        'host': host,
                        'cve': cve,
                        'repetitions': cycles,
```

```
                        'timeline': cve_timeline
                    })

        return {
            'pattern': 'REPETITION_COMPULSION',
            'score': min(repetition_score * 0.1, 1.0),
            'evidence': evidence,
            'cpf_category': '[8.3]'
        }

    def detect_temporal_patterns(self):
        """
        Detect: Time-based vulnerability patterns
        """
        # Convert to datetime for analysis
        self.data['detection_hour'] =
pd.to_datetime(self.data['first_detected']).dt.hour
        self.data['detection_day'] =
pd.to_datetime(self.data['first_detected']).dt.dayofweek

        # Friday afternoon pattern
        friday_afternoon = self.data[(self.data['detection_day'] == 4) &
(self.data['detection_hour'] >= 14)]
        other_times = self.data[~((self.data['detection_day'] == 4) &
(self.data['detection_hour'] >= 14))]

        friday_patch_rate = len(friday_afternoon[friday_afternoon['status'] ==
'RESOLVED']) / len(friday_afternoon)
        other_patch_rate = len(other_times[other_times['status'] == 'RESOLVED']) /
len(other_times)

        temporal_score = max(0, (other_patch_rate - friday_patch_rate))

        return {
            'pattern': 'TEMPORAL_VULNERABILITY',
            'score': temporal_score,
            'evidence': {
                'friday_afternoon_patch_rate': friday_patch_rate,
                'other_times_patch_rate': other_patch_rate,
                'vulnerability_window': 'Friday 14:00-18:00'
            },
            'cpf_category': '[2.7]'
        }

    def detect_cognitive_overload(self):
        """
        Detect: Overwhelming number of vulnerabilities causing paralysis
        """
        # Calculate vulnerabilities per host
```

```
        vuln_counts = self.data.groupby('host_key')['cve'].count()

        # Hosts with >100 vulnerabilities
        overloaded_hosts = vuln_counts[vuln_counts > 100]

        # Check patch rate for overloaded vs normal hosts
        overloaded_patch_rate = self.calculate_patch_rate(overloaded_hosts.index)
        normal_patch_rate = self.calculate_patch_rate(vuln_counts[vuln_counts <=
100].index)

        overload_score = max(0, (normal_patch_rate - overloaded_patch_rate))

        return {
            'pattern': 'COGNITIVE_OVERLOAD',
            'score': overload_score,
            'evidence': {
                'overloaded_hosts': len(overloaded_hosts),
                'avg_vulns_per_overloaded': overloaded_hosts.mean(),
                'overloaded_patch_rate': overloaded_patch_rate,
                'normal_patch_rate': normal_patch_rate
            },
            'cpf_category': '[5.3]'
        }

    def classify_host(self, host_key):
        """Classify host type based on naming convention"""
        if 'exec' in host_key or 'ceo' in host_key or 'cfo' in host_key:
            return 'executive'
        elif 'prod' in host_key:
            return 'production'
        elif 'dev' in host_key:
            return 'development'
        elif 'it' in host_key or 'admin' in host_key:
            return 'it_infrastructure'
        return 'general'
```

Step 4: CPF Score Calculation

```
class CPFScoreCalculator:
    def __init__(self, pattern_results):
        self.patterns = pattern_results
        self.category_weights = {
            '[1.x]': 0.12,  # Authority
            '[2.x]': 0.10,  # Temporal
            '[3.x]': 0.08,  # Social
            '[4.x]': 0.11,  # Affective
            '[5.x]': 0.13,  # Cognitive
            '[6.x]': 0.09,  # Group
```

```
            '[7.x]': 0.10,  # Stress
            '[8.x]': 0.15,  # Unconscious
            '[9.x]': 0.07,  # AI
            '[10.x]': 0.05  # Convergent
        }

    def calculate_scores(self):
        # Initialize category scores
        category_scores = {cat: 0.0 for cat in self.category_weights.keys()}

        # Aggregate pattern scores by category
        for pattern in self.patterns:
            category = pattern['cpf_category'].split(']')[0] + ']'
            if category in category_scores:
                category_scores[category] = max(category_scores[category],
pattern['score'])

        # Calculate weighted total
        total_score = sum(category_scores[cat] * self.category_weights[cat]
                          for cat in category_scores)

        # Identify convergent risks
        high_risk_patterns = [p for p in self.patterns if p['score'] > 0.7]
        convergent_risk = len(high_risk_patterns) >= 3

        return {
            'total_cpf_score': total_score,
            'category_scores': category_scores,
            'convergent_risk': convergent_risk,
            'high_risk_patterns': high_risk_patterns
        }
```

Step 5: Priority Adjustment Engine

```
class PriorityAdjuster:
    def __init__(self, vulnerabilities, cpf_scores):
        self.vulns = vulnerabilities
        self.cpf = cpf_scores

    def adjust_priorities(self):
        adjusted = []

        for _, vuln in self.vulns.iterrows():
            base_priority = self.calculate_base_priority(vuln)
            psychological_multiplier = self.calculate_psychological_multiplier(vuln)

            adjusted.append({
                'cve': vuln['cve'],
```

```
                'host': vuln['host_key'],
                'original_priority': base_priority,
                'psychological_multiplier': psychological_multiplier,
                'adjusted_priority': base_priority * psychological_multiplier,
                'action': self.determine_action(base_priority *
psychological_multiplier)
            })

        # Sort by adjusted priority
        adjusted.sort(key=lambda x: x['adjusted_priority'], reverse=True)
        return adjusted

    def calculate_base_priority(self, vuln):
        # CVSS-like calculation
        severity_scores = {'Critical': 10, 'High': 7, 'Medium': 4, 'Low': 1}
        base = severity_scores.get(vuln['severity'], 1)

        # Adjust for exploit availability
        if vuln.get('exploit_available'):
            base *= 1.5
        if vuln.get('malware_kits', 0) > 0:
            base *= 2.0

        return base

    def calculate_psychological_multiplier(self, vuln):
        multiplier = 1.0

        # Check if CVE appears in pattern evidence
        for pattern in self.cpf['high_risk_patterns']:
            if pattern['pattern'] == 'REPETITION_COMPULSION':
                # CVEs that repeat get highest priority
                for evidence in pattern['evidence']:
                    if vuln['cve'] == evidence['cve']:
                        multiplier = max(multiplier, 3.0)

            elif pattern['pattern'] == 'SPLITTING':
                # CVEs on "good objects" get boosted
                for evidence in pattern['evidence']:
                    if vuln['cve'] == evidence['cve'] and \
                       vuln['host_type'] == evidence['good_object']:
                        multiplier = max(multiplier, 2.5)

            elif pattern['pattern'] == 'MANIC_DEFENSE':
                # CVEs without PoC get boosted if manic defense active
                if not vuln.get('exploit_available'):
                    multiplier = max(multiplier, 2.0)

        # Convergent risk multiplier
```

```
        if self.cpf['convergent_risk']:
            multiplier *= 1.5

        return multiplier

    def determine_action(self, priority):
        if priority > 30:
            return 'EMERGENCY: Patch within 24 hours'
        elif priority > 20:
            return 'CRITICAL: Patch within 72 hours'
        elif priority > 10:
            return 'HIGH: Patch within 1 week'
        elif priority > 5:
            return 'MEDIUM: Patch within 1 month'
        return 'LOW: Schedule for regular maintenance'
```

Step 6: Real-time Monitoring

```
class CPFMonitor:
    def __init__(self, config):
        self.qualys = QualysExtractor(config['qualys'])
        self.tenable = TenableExtractor(config['tenable'])
        self.rapid7 = Rapid7Extractor(config['rapid7'])
        self.last_check = datetime.now()

    def continuous_monitoring(self, interval_minutes=60):
        while True:
            try:
                # Pull latest data
                current_data = self.get_current_state()

                # Detect patterns
                patterns = self.detect_all_patterns(current_data)

                # Calculate CPF scores
                cpf_scores = CPFScoreCalculator(patterns).calculate_scores()

                # Check alert conditions
                self.check_alert_conditions(cpf_scores, patterns)

                # Adjust priorities
                adjusted = PriorityAdjuster(current_data,
cpf_scores).adjust_priorities()

                # Push to dashboard
                self.update_dashboard(cpf_scores, adjusted)

                # Wait for next cycle
```

```
                time.sleep(interval_minutes * 60)

            except Exception as e:
                self.log_error(e)
                time.sleep(300)  # Wait 5 minutes on error

    def check_alert_conditions(self, cpf_scores, patterns):
        alerts = []

        # Critical convergent risk
        if cpf_scores['convergent_risk'] and cpf_scores['total_cpf_score'] > 0.7:
            alerts.append({
                'level': 'CRITICAL',
                'message': 'Multiple psychological vulnerabilities converging - breach
imminent',
                'action': 'Immediate intervention required'
            })

        # Pattern-specific alerts
        for pattern in patterns:
            if pattern['pattern'] == 'REPETITION_COMPULSION' and pattern['score'] >
0.5:
                alerts.append({
                    'level': 'HIGH',
                    'message': f'Repetition pattern detected for
{len(pattern["evidence"])} CVEs',
                    'action': 'Address organizational trauma before next cycle'
                })

            elif pattern['pattern'] == 'MANIC_DEFENSE' and pattern['score'] > 0.6:
                alerts.append({
                    'level': 'HIGH',
                    'message': 'Organization in manic defense - ignoring non-public
threats',
                    'action': 'Priority boost for CVEs without PoC'
                })

        # Friday afternoon check
        if datetime.now().weekday() == 4 and datetime.now().hour >= 14:
            if any(p['pattern'] == 'TEMPORAL_VULNERABILITY' for p in patterns):
                alerts.append({
                    'level': 'MEDIUM',
                    'message': 'Entering Friday vulnerability window',
                    'action': 'Increase monitoring for social engineering'
                })

        # Send alerts
        for alert in alerts:
            self.send_alert(alert)
```

Step 7: Output Integration

```
class Dashboard:
    def __init__(self):
        self.api_endpoint = "https://dashboard.internal/api/cpf"

    def format_output(self, cpf_scores, adjusted_priorities, patterns):
        output = {
            'timestamp': datetime.now().isoformat(),
            'cpf_metrics': {
                'total_score': cpf_scores['total_cpf_score'],
                'category_breakdown': cpf_scores['category_scores'],
                'risk_level': self.calculate_risk_level(cpf_scores['total_cpf_score']),
                'convergent_risk': cpf_scores['convergent_risk']
            },
            'detected_patterns': [
                {
                    'name': p['pattern'],
                    'severity': self.score_to_severity(p['score']),
                    'score': p['score'],
                    'evidence_count': len(p.get('evidence', [])),
                    'category': p['cpf_category']
                }
                for p in patterns
            ],
            'priority_queue': adjusted_priorities[:20],  # Top 20
            'predictions': {
                'breach_probability_30d':
self.calculate_breach_probability(cpf_scores),
                'most_likely_vector': self.predict_attack_vector(patterns),
                'vulnerability_windows': self.identify_windows(patterns)
            },
            'recommendations': self.generate_recommendations(patterns, cpf_scores)
        }

        return output

    def push_to_dashboard(self, data):
        response = requests.post(
            self.api_endpoint,
            json=data,
            headers={'Content-Type': 'application/json'}
        )
        return response.status_code == 200

    def calculate_risk_level(self, score):
        if score > 0.7: return 'CRITICAL'
        elif score > 0.5: return 'HIGH'
        elif score > 0.3: return 'MEDIUM'
```

```
        return 'LOW'

    def score_to_severity(self, score):
        if score > 0.7: return 'RED'
        elif score > 0.4: return 'YELLOW'
        return 'GREEN'
```

Complete Integration Flow

```
def main():
    # Load configuration
    config = load_config('config.yaml')

    # Step 1: Extract data from all three scanners
    qualys = QualysExtractor(config['qualys'])
    tenable = TenableExtractor(config['tenable'])
    rapid7 = Rapid7Extractor(config['rapid7'])

    qualys_data = qualys.get_vulnerability_data()
    tenable_data = tenable.get_vulnerability_data()
    rapid7_data = rapid7.get_vulnerability_data()

    # Step 2: Normalize and merge
    normalizer = DataNormalizer()
    consolidated = normalizer.merge_scanner_data(qualys_data, tenable_data,
rapid7_data)

    # Step 3: Detect patterns
    engine = CPFPatternEngine(consolidated)
    patterns = [
        engine.detect_manic_defense(),
        engine.detect_splitting(),
        engine.detect_repetition_compulsion(),
        engine.detect_temporal_patterns(),
        engine.detect_cognitive_overload()
    ]

    # Step 4: Calculate CPF scores
    calculator = CPFScoreCalculator(patterns)
    cpf_scores = calculator.calculate_scores()

    # Step 5: Adjust priorities
    adjuster = PriorityAdjuster(consolidated, cpf_scores)
    adjusted_priorities = adjuster.adjust_priorities()

    # Step 6: Format and push to dashboard
    dashboard = Dashboard()
    output = dashboard.format_output(cpf_scores, adjusted_priorities, patterns)
```

```
    dashboard.push_to_dashboard(output)

    # Step 7: Start continuous monitoring
    monitor = CPFMonitor(config)
    monitor.continuous_monitoring(interval_minutes=60)

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