CPF Implementation Guide: From Vulnerability Scanner Data to Psychological Scores

Data Sources from Qualys/Tenable/Rapid7

Available Data Structures

```python
# Qualys/Tenable/Rapid7 typical data structure
vulnerability_data = {
    "host_id": "srv-prod-001",
    "scan_date": "2025-08-29T14:30:00Z",
    "confirmed_vulns": [
        {
            "cve": "CVE-2024-1234",
            "cvss": 9.8,
            "discovered_date": "2025-01-15",
            "patch_available_date": "2025-01-20",
            "exploitability": "HIGH",
            "patch_status": "MISSING"
        }
    ],
    "potential_vulns": [
        {
            "cve": "CVE-2024-5678",
            "confidence": 75,
            "reason": "Version fingerprinting uncertain"
        }
    ],
    "installed_software": [
        {
            "name": "Apache",
            "version": "2.2.15",
            "install_date": "2019-03-15",
            "authorized": False
        }
    ],
    "running_processes": [
        {
            "name": "httpd",
            "pid": 1234,
            "user": "root",
            "start_time": "2025-08-29T03:00:00Z",
            "cpu_usage": 45.2
        }
    ],
    "users": [
        {
            "username": "admin",
            "last_login": "2025-08-29T18:45:00Z",
            "privilege_level": "administrator",
            "login_frequency": "daily"
        }
    ],
    "enrichment": {
        "poc_published": ["CVE-2024-1234"],
        "news_mentions": ["CVE-2024-1234"],
        "github_activity": "HIGH",
        "exploit_kits": ["CVE-2024-1234"]
    }
}
```

Pattern Detection Implementation

PATTERN 1: Manic Defense Detection

```
class ManicDefenseDetector:
    """
    Detects manic defense through patch behavior around PoC publication
    Data sources: Qualys/Tenable vulnerability data + enrichment
    """

    def __init__(self, vuln_history):
        self.vuln_history = vuln_history
        self.threshold_days_before_poc = 90
        self.threshold_hours_after_poc = 48

    def detect_pattern(self):
        pattern_score = 0
        manic_events = []

        for vuln in self.vuln_history:
            if vuln['cve'] in vuln['enrichment']['poc_published']:
                # Calculate time between CVE discovery and patch
                time_before_poc = days_between(
                    vuln['discovered_date'],
                    vuln['poc_publish_date']
                )
                time_after_poc = hours_between(
                    vuln['poc_publish_date'],
                    vuln['patch_applied_date']
                )
                if (time_before_poc > self.threshold_days_before_poc and
                    time_after_poc < self.threshold_hours_after_poc):
                    # Manic defense pattern detected
                    pattern_score += 1
                    manic_events.append({
                        'cve': vuln['cve'],
                        'ignored_days': time_before_poc,
                        'panic_hours': time_after_poc
                    })

        return {
            'pattern': 'MANIC_DEFENSE',
            'cpf_category': '[8.6] Defense mechanism interference',
            'score': min(pattern_score * 0.2, 1.0),  # Normalize to 0-1
            'severity': self.calculate_severity(pattern_score),
            'evidence': manic_events,
            'prediction': 'Organization vulnerable to 0-day attacks',
            'intervention': 'Address omnipotent fantasies about security'
        }

    def calculate_severity(self, score):
        if score >= 5: return 'RED'
        elif score >= 2: return 'YELLOW'
        return 'GREEN'
```

PATTERN 2: Splitting Detection

```
class SplittingDetector:
    """
    Detects splitting through differential treatment of identical CVEs
    Data source: Qualys/Tenable host-level vulnerability data
    """

    def __init__(self, fleet_data):
        self.fleet_data = fleet_data  # All hosts data

    def detect_pattern(self):
        # Group hosts by characteristics
        host_groups = {
            'executive': [],
            'production': [],
            'development': [],
            'IT': []
        }

        # Classify hosts based on naming/user patterns
        for host in self.fleet_data:
            host_type = self.classify_host(host)
            host_groups[host_type].append(host)

        # Find CVEs that exist across multiple groups
        common_cves = self.find_common_cves(host_groups)

        splitting_score = 0
        splitting_evidence = []

        for cve in common_cves:
            patch_rates = {}
            for group_name, hosts in host_groups.items():
                patch_rates[group_name] = self.calculate_patch_rate(hosts, cve)

            # Detect splitting: same CVE, vastly different treatment
            if self.is_splitting_pattern(patch_rates):
                splitting_score += 1
                splitting_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',
            'cpf_category': '[4.9] Object relations splitting',
            'score': min(splitting_score * 0.15, 1.0),
            'severity': self.calculate_severity(splitting_score),
            'evidence': splitting_evidence,
            'prediction': f'Breach via {splitting_evidence[0]["good_object"]} systems',
            'intervention': 'Workshop on whole-object relations'
        }

    def is_splitting_pattern(self, patch_rates):
        values = list(patch_rates.values())
        return max(values) - min(values) > 0.7  # 70% difference
```

PATTERN 3: Repetition Compulsion Detection

```
class RepetitionCompulsionDetector:
    """
    Detects CVEs that keep returning despite patching
    Data source: Tenable/Qualys historical scan data
    """

    def __init__(self, scan_history):
        self.scan_history = scan_history  # Multiple scans over time
        self.min_repetitions = 3
    def detect_pattern(self):
        cve_timeline = self.build_cve_timeline()
        repetition_score = 0
        compulsive_cves = []

        for cve, timeline in cve_timeline.items():
            repetitions = self.count_repetitions(timeline)

            if repetitions >= self.min_repetitions:
                repetition_score += repetitions
                compulsive_cves.append({
                    'cve': cve,
                    'repetitions': repetitions,
                    'pattern': timeline,
                    'trauma_category': self.identify_trauma_type(cve)
                })

        return {
            'pattern': 'REPETITION_COMPULSION',
            'cpf_category': '[8.3] Repetition compulsion patterns',
            'score': min(repetition_score * 0.1, 1.0),
            'severity': 'RED' if repetition_score > 0 else 'GREEN',
            'evidence': compulsive_cves,
            'prediction': f'{compulsive_cves[0]["cve"]} will be breach vector',
            'intervention': 'Identify organizational trauma around this CVE type'
        }

    def count_repetitions(self, timeline):
        # Count patch->reappear cycles
        repetitions = 0
        for i in range(len(timeline) - 1):
            if timeline[i] == 'PATCHED' and timeline[i+1] == 'VULNERABLE':
                repetitions += 1
        return repetitions
```

PATTERN 4: Temporal Vulnerability Windows

```
class TemporalVulnerabilityDetector:
    """
    Detects time-based vulnerability patterns
    Data source: Rapid7 Nexpose scan timestamps + patch history
    """

    def __init__(self, temporal_data):
        self.temporal_data = temporal_data

    def detect_pattern(self):
        patterns = {
            'friday_fade': self.detect_friday_fade(),
            'holiday_gaps': self.detect_holiday_gaps(),
            'audit_theater': self.detect_audit_cycles(),
            'ego_depletion': self.detect_progressive_decay()
        }

        # Combine temporal patterns for overall score
        temporal_score = sum(p['score'] for p in patterns.values()) / len(patterns)

        return {
            'pattern': 'TEMPORAL_VULNERABILITY',
            'cpf_category': '[2.x] Temporal Vulnerabilities',
            'score': temporal_score,
            'severity': self.calculate_severity(temporal_score),
            'sub_patterns': patterns,
            'prediction': self.predict_vulnerability_window(patterns),
            'intervention': 'Implement psychological support during high-risk periods'
        }

    def detect_friday_fade(self):
        friday_patches = []
        other_day_patches = []

        for patch in self.temporal_data['patch_history']:
            patch_day = patch['timestamp'].weekday()
            success_rate = patch['success_rate']

            if patch_day == 4:  # Friday
                friday_patches.append(success_rate)
            else:
                other_day_patches.append(success_rate)

        friday_avg = sum(friday_patches) / len(friday_patches) if friday_patches else 0
        other_avg = sum(other_day_patches) / len(other_day_patches) if
other_day_patches else 0

        fade_score = max(0, (other_avg - friday_avg) / other_avg)

        return {
            'score': fade_score,
            'friday_success_rate': friday_avg,
            'other_days_rate': other_avg,
            'interpretation': 'Superego dissolution in liminal time'
        }

    def predict_vulnerability_window(self, patterns):
        if patterns['friday_fade']['score'] > 0.3:
            return "Maximum vulnerability: Friday 14:00-17:00"
        elif patterns['holiday_gaps']['score'] > 0.5:
            return "Critical exposure during next holiday period"
        return "No significant temporal vulnerability detected"
```

PATTERN 5: Cognitive Overload Detection

```
class CognitiveOverloadDetector:
    """
    Detects cognitive overload from vulnerability volume and response patterns
    Data source: Qualys VMDR aggregated metrics
    """

    def __init__(self, workload_data):
        self.workload_data = workload_data

    def detect_pattern(self):
        metrics = {
            'alert_fatigue': self.calculate_alert_fatigue(),
            'decision_paralysis': self.calculate_decision_paralysis(),
            'complexity_score': self.calculate_complexity_overload()
        }

        overload_score = self.aggregate_overload_score(metrics)

        return {
            'pattern': 'COGNITIVE_OVERLOAD',
            'cpf_category': '[5.x] Cognitive Overload Vulnerabilities',
            'score': overload_score,
            'severity': self.calculate_severity(overload_score),
            'metrics': metrics,
            'prediction': 'Critical CVEs ignored due to overload',
            'intervention': 'Reduce cognitive load before adding more tools'
        }

    def calculate_alert_fatigue(self):
        # Measure response rate degradation over time
        weekly_response_rates = []

        for week in self.workload_data['weekly_metrics']:
            total_alerts = week['total_alerts']
            investigated = week['alerts_investigated']
            response_rate = investigated / total_alerts if total_alerts > 0 else 0
            weekly_response_rates.append(response_rate)

        # Calculate degradation slope
        if len(weekly_response_rates) > 4:
            early_avg = sum(weekly_response_rates[:4]) / 4
            recent_avg = sum(weekly_response_rates[-4:]) / 4
            fatigue_score = max(0, (early_avg - recent_avg) / early_avg)
        else:
            fatigue_score = 0

        return {
            'score': fatigue_score,
            'current_response_rate': weekly_response_rates[-1] if weekly_response_rates
else 0,
            'interpretation': 'Progressive alert desensitization'
        }
```

PATTERN 6: Shadow IT Detection

```
class ShadowITDetector:
    """
    Detects unauthorized software patterns indicating group dynamics
    Data source: Tenable/Qualys software inventory
    """

    def __init__(self, software_inventory):
        self.software_inventory = software_inventory
        self.authorized_list = self.load_authorized_software()

    def detect_pattern(self):
        shadow_it_map = {}

        for host in self.software_inventory:
            department = self.get_department(host)
            unauthorized = self.find_unauthorized_software(host)

            if department not in shadow_it_map:
                shadow_it_map[department] = []
            shadow_it_map[department].extend(unauthorized)

        # Analyze clustering patterns
        shadow_patterns = []
        for dept, software_list in shadow_it_map.items():
            if len(software_list) > 10:  # Significant shadow IT
                shadow_patterns.append({
                    'department': dept,
                    'unauthorized_count': len(software_list),
                    'common_software': self.find_common_patterns(software_list),
                    'group_dynamic': 'Fight-flight against IT authority'
                })

        shadow_score = len(shadow_patterns) * 0.2

        return {
            'pattern': 'SHADOW_IT',
            'cpf_category': '[6.7] Fight-flight security postures',
            'score': min(shadow_score, 1.0),
            'severity': self.calculate_severity(shadow_score),
            'evidence': shadow_patterns,
            'prediction': 'Ransomware entry via unauthorized SaaS/tools',
            'intervention': 'Address departmental rebellion against IT'
        }
```

Aggregated CPF Scoring System

```
class CPFScoreCalculator:
    """
    Aggregates all pattern detections into unified CPF score
    Integrates with Qualys/Tenable/Rapid7 APIs
    """

    def __init__(self, scanner_api):
        self.scanner_api = scanner_api  # Qualys/Tenable/Rapid7 API client
        self.detectors = [
            ManicDefenseDetector,
            SplittingDetector,
            RepetitionCompulsionDetector,
            TemporalVulnerabilityDetector,
            CognitiveOverloadDetector,
            ShadowITDetector
        ]

    def calculate_cpf_score(self, customer_id):
        # Pull data from scanner API
        raw_data = self.fetch_scanner_data(customer_id)

        # Run all pattern detectors
        pattern_results = []
        for detector_class in self.detectors:
            detector = detector_class(raw_data)
            result = detector.detect_pattern()
            pattern_results.append(result)

        # Calculate aggregate scores
        cpf_scores = self.aggregate_scores(pattern_results)

        # Identify convergent risks (multiple red flags)
        convergent_risk = self.calculate_convergent_risk(pattern_results)

        return {
            'customer_id': customer_id,
            'scan_date': datetime.now().isoformat(),
            'cpf_total_score': cpf_scores['total'],
            'category_scores': cpf_scores['by_category'],
            'detected_patterns': pattern_results,
            'convergent_risk': convergent_risk,
            'priority_interventions': self.prioritize_interventions(pattern_results),
            'predicted_breach_vectors': self.aggregate_predictions(pattern_results)
        }

    def fetch_scanner_data(self, customer_id):
        """
        Fetch data from Qualys/Tenable/Rapid7 APIs
        """
        data = {
            'vulnerability_data': self.scanner_api.get_vulnerabilities(customer_id),
            'asset_inventory': self.scanner_api.get_assets(customer_id),
            'scan_history': self.scanner_api.get_scan_history(customer_id, days=180),
            'software_inventory': self.scanner_api.get_software(customer_id),
            'process_snapshots': self.scanner_api.get_processes(customer_id),
            'enrichment': self.fetch_threat_intelligence()
        }
        return data

    def aggregate_scores(self, pattern_results):
        # CPF scoring by category
        category_scores = {
            '[1.x]': 0,  # Authority
            '[2.x]': 0,  # Temporal
            '[3.x]': 0,  # Social
            '[4.x]': 0,  # Affective
            '[5.x]': 0,  # Cognitive
            '[6.x]': 0,  # Group
            '[7.x]': 0,  # Stress
            '[8.x]': 0,  # Unconscious
            '[9.x]': 0,  # AI
            '[10.x]': 0  # Convergent
        }

        for result in pattern_results:
            category = result['cpf_category'].split(']')[0] + ']'
            category_scores[category] = max(
                category_scores[category],
                result['score']
            )

        total_score = sum(category_scores.values()) / len(category_scores)

        return {
            'total': total_score,
            'by_category': category_scores
        }

    def calculate_convergent_risk(self, pattern_results):
        """
        Identify when multiple psychological states converge
        creating perfect storm conditions
        """
        red_patterns = [p for p in pattern_results if p['severity'] == 'RED']

        if len(red_patterns) >= 3:
            return {
                'level': 'CRITICAL',
                'converging_patterns': [p['pattern'] for p in red_patterns],
                'prediction': 'Breach imminent within 30 days',
                'cpf_category': '[10.1] Perfect storm conditions'
            }
        elif len(red_patterns) >= 2:
            return {
                'level': 'HIGH',
                'converging_patterns': [p['pattern'] for p in red_patterns],
                'prediction': 'Elevated risk of successful attack',
                'cpf_category': '[10.4] Swiss cheese alignment'
            }
        return {
            'level': 'NORMAL',
            'prediction': 'No convergent risks detected'
        }
```

Integration with Infrastructure

```
class CPFIntegration:
    """
    Production integration for vulnerability management module
    """

    def __init__(self, config):
        self.qualys_api = QualysAPI(config['qualys'])
        self.tenable_api = TenableAPI(config['tenable'])
        self.rapid7_api = Rapid7API(config['rapid7'])
        self.cpf_calculator = CPFScoreCalculator()

    def process_customer(self, customer_id):
        # Aggregate data from all scanners
        combined_data = self.aggregate_scanner_data(customer_id)

        # Calculate CPF scores
        cpf_results = self.cpf_calculator.calculate_cpf_score(combined_data)
```

```python
        # Adjust CVE priorities based on psychological state
        adjusted_priorities = self.adjust_cve_priorities(
            combined_data['vulnerabilities'],
            cpf_results
        )

        # Generate actionable report
        report = self.generate_cpf_report(cpf_results, adjusted_priorities)

        return report

    def adjust_cve_priorities(self, vulnerabilities, cpf_results):
        """
        Re-prioritize CVEs based on psychological vulnerabilities
        """
        adjusted = []

        for vuln in vulnerabilities:
            base_score = vuln['cvss']
            psychological_multiplier = 1.0

            # Apply psychological adjustments
            for pattern in cpf_results['detected_patterns']:
                if pattern['pattern'] == 'REPETITION_COMPULSION':
                    # CVEs that match repetition pattern get highest priority
                    if vuln['cve'] in pattern['evidence']:
                        psychological_multiplier = 2.5

                elif pattern['pattern'] == 'SPLITTING':
                    # CVEs on "good object" systems get boosted
                    if vuln['host_type'] in pattern['evidence']['good_object']:
                        psychological_multiplier = 2.0

                elif pattern['pattern'] == 'MANIC_DEFENSE':
                    # CVEs without PoC get boosted if manic defense detected
                    if vuln['cve'] not in vuln['enrichment']['poc_published']:
                        psychological_multiplier = 1.8

            adjusted.append({
                'cve': vuln['cve'],
                'original_priority': base_score,
                'cpf_adjusted_priority': base_score * psychological_multiplier,
                'psychological_factor': psychological_multiplier,
                'reasoning': self.explain_adjustment(pattern, vuln)
            })

        return sorted(adjusted, key=lambda x: x['cpf_adjusted_priority'], reverse=True)
```

Real-time Monitoring Rules

```python
class CPFMonitoringRules:
    """
    Real-time alerting based on psychological state changes
    """

    def __init__(self, alert_system):
        self.alert_system = alert_system
        self.rules = self.define_rules()

    def define_rules(self):
        return [
            {
                'name': 'Manic Defense Collapse',
                'condition': lambda data: (
                    data['poc_published_last_24h'] and
                    data['unpatched_critical_cves'] > 50
                ),
                'alert': 'CRITICAL: Manic defense collapse imminent. Expect panic patching.',
                'action': 'Pre-position support for emergency patching'
            },
            {
                'name': 'Friday Vulnerability Window',
                'condition': lambda data: (
                    datetime.now().weekday() == 4 and
                    datetime.now().hour >= 14 and
                    data['cpf_scores']['temporal'] > 0.7
                ),
                'alert': 'HIGH: Entering Friday fade window with high temporal vulnerability',
                'action': 'Increase SOC monitoring for next 4 hours'
            },
            {
                'name': 'Repetition Compulsion Active',
                'condition': lambda data: (
                    data['recurring_cve_count'] > 0 and
                    data['days_since_last_recurrence'] > 85
                ),
                'alert': 'MEDIUM: Repetition cycle approaching for recurring CVEs',
                'action': 'Schedule intervention before day 90'
            },
            {
                'name': 'Cognitive Overload Crisis',
                'condition': lambda data: (
                    data['alert_response_rate'] < 0.1 and
                    data['new_cves_per_day'] > 100
                ),
                'alert': 'CRITICAL: Team in cognitive overload, security blindness active',
                'action': 'Immediate workload reduction required'
            }
        ]

    def evaluate_rules(self, current_data):
        triggered_alerts = []

        for rule in self.rules:
            if rule['condition'](current_data):
                self.alert_system.send_alert(
                    level=rule['alert'].split(':')[0],
                    message=rule['alert'],
                    recommended_action=rule['action']
                )
                triggered_alerts.append(rule['name'])

        return triggered_alerts
```

Output Format for Dashboard

```
def generate_dashboard_json(cpf_results):
    """
    Format CPF results for customer dashboard
    """
    return {
        "metadata": {
            "customer_id": cpf_results['customer_id'],
            "scan_date": cpf_results['scan_date'],
            "data_sources": ["Qualys", "Tenable", "Rapid7"],
            "cpf_version": "1.0"
        },
        "executive_summary": {
            "overall_psychological_health": cpf_results['cpf_total_score'],
            "risk_level": calculate_risk_level(cpf_results['cpf_total_score']),
            "dominant_vulnerability": cpf_results['detected_patterns'][0]['pattern'],
            "breach_prediction": cpf_results['predicted_breach_vectors'][0]
        },
        "psychological_state": {
            "active_patterns": [
                {
                    "pattern": p['pattern'],
                    "severity": p['severity'],
                    "description": p['interpretation'],
                    "evidence_count": len(p['evidence'])
                }
                for p in cpf_results['detected_patterns']
            ],
            "convergent_risk": cpf_results['convergent_risk']
        },
        "adjusted_priorities": {
            "critical_cves": [
                {
                    "cve": cve['cve'],
                    "traditional_score": cve['original_priority'],
                    "cpf_adjusted_score": cve['cpf_adjusted_priority'],
                    "reason": cve['reasoning'],
                    "action": "PATCH IMMEDIATELY"
                }
                for cve in cpf_results['adjusted_priorities'][:10]
            ]
        },
        "predictions": {
            "vulnerability_windows": [
                {
                    "timeframe": "Friday 14:00-17:00",
                    "risk_multiplier": 3.2,
                    "attack_type": "Phishing/Social Engineering"
                }
            ],
            "likely_breach_vector": cpf_results['predicted_breach_vectors'][0],
            "timeline": "Next 30-60 days based on pattern convergence"
        },
        "recommendations": {
            "immediate": cpf_results['priority_interventions'][:3],
            "medium_term": cpf_results['priority_interventions'][3:6],
            "strategic": [
                "Address underlying psychological dynamics",
                "Implement CPF-aware security training",
                "Regular psychological state assessments"
            ]
        }
    }
```