Production Security Plan - 7 Day Sprint

RichesReach - Stop-Ship Items → Production-Ready

🚨 STOP-SHIP ITEMS (Fix First - Days 1-2)

1. Hardcoded Secrets & Passwords

Status: 🔴 CRITICAL - Found in codebase

Action Items: - [ ] Run secret scan: `rg -n "test123|password\s*=\s*['\"]|SECRET|API_KEY" .` - [ ] Replace all hardcoded values with environment variables - [ ] Add pre-commit hook to block secret commits - [ ] Add CI secret scanning (truffleHog/git-secrets)

Files to Fix: - `deployment_package/backend/core/banking_views.py` (test passwords) - `deployment_package/backend/core/enhanced_api_service.py` (API keys) - Any other files found in scan

Code Pattern:

```
# ❌ BAD
password='test123'
api_key = "K0A7XYLDNXHNQ1WI"

# ✅ GOOD
password = os.getenv('TEST_USER_PASSWORD', '')  # Empty in production
api_key = os.getenv('ALPHA_VANTAGE_API_KEY') or get_secret_from_manager('alpha_vanta
```

2. SSL Verification Disabled

Status: 🔴 CRITICAL - Found in dev code

Action Items: - [ ] Find all `verify=False` and `CERT_NONE` instances - [ ] Add production guard: `verify=os.getenv('SSL_VERIFY', 'true').lower() == 'true'` - [ ] Ensure `SSL_VERIFY=true` in production environment - [ ] Remove or guard all `CERT_NONE` code paths

Files to Fix: - `deployment_package/backend/main.py` (SSL context) - Any other files with SSL verification disabled

Code Pattern:

```
# ❌ BAD
ssl_context.verify_mode = ssl.CERT_NONE
requests.get(url, verify=False)

# ✅ GOOD
import os
verify_ssl = os.getenv('SSL_VERIFY', 'true').lower() == 'true'
if not verify_ssl and os.getenv('ENVIRONMENT') == 'production':
    raise ValueError("SSL verification cannot be disabled in production")
requests.get(url, verify=verify_ssl)
```

3. CSRF Protection Strategy

Status: ⚠️ NEEDS VERIFICATION

Action Items: - [ ] Verify all API endpoints use Bearer token auth (not cookies) - [ ] Confirm Django session middleware is disabled for API views - [ ] Document CSRF strategy in code comments - [ ] If cookie-based auth exists, add CSRF protection or separate domains

Verification Checklist: - [ ] Mobile app sends `Authorization: Bearer <token>`? - [ ] Web app (if any) uses Bearer tokens or separate API domain? - [ ] No cookie-based sessions for API endpoints? - [ ] GraphQL endpoint uses Bearer tokens?

If Bearer tokens only:

```
 # ✅ SAFE - Document why CSRF exempt is OK
@method_decorator(csrf_exempt, name='dispatch')  # Safe: Bearer token auth only
class BankingView(View):
    """
    CSRF exempt because:
    1. All requests use Authorization: Bearer <token>
    2. No cookie-based sessions
    3. Stateless API design
    """
```

If cookies exist:

```
 # ✅ NEEDS CSRF
from django.views.decorators.csrf import csrf_protect
@method_decorator(csrf_protect, name='dispatch')
class WebView(View):
    # Cookie-based auth requires CSRF protection
```

---

4. API Keys in Code → Secrets Manager

Status: 🔴 CRITICAL - Found hardcoded keys

Action Items: - [ ] Move all API keys to AWS Secrets Manager - [ ] Create secrets manager integration helper - [ ] Update all services to load from secrets manager - [ ] Document rotation process

Files to Fix: - `deployment_package/backend/core/enhanced_api_service.py` - All services loading API keys from env (move to secrets manager)

Code Pattern:

```
 # ❌ BAD
self.api_keys = ["K0A7XYLDNXHNQ1WI", "OHYSFF1AE446O7CR"]
```

```
# ✅ GOOD
from .secrets_manager import get_secret

def __init__(self):
    self.api_keys = [
        get_secret('alpha_vantage_key_1'),
        get_secret('alpha_vantage_key_2'),
    ]
```

---

5. Raw SQL Audit

Status: ⚠️ NEEDS AUDIT

Action Items: - [ ] Find all raw SQL queries: `rg -n "cursor\.execute" .` - [ ] Verify all use parameterized queries (no string formatting) - [ ] Replace with Django ORM where possible - [ ] Add SQL injection tests

Code Pattern:

```
 # ❌ BAD - SQL Injection Risk
cursor.execute(f"SELECT * FROM users WHERE email = '{email}'")

# ✅ GOOD - Parameterized
cursor.execute("SELECT * FROM users WHERE email = %s", [email])

# ✅ BEST - Django ORM
User.objects.filter(email=email)
```

📅 7-DAY ACTION PLAN

Day 1-2: Stop-Ship Items

Goal: Remove all hardcoded secrets, lock down SSL verification

Tasks: 1. Run secret scan and create fix list 2. Replace all hardcoded passwords with env vars 3. Add production guard for SSL verification 4. Add pre-commit hook for secret detection 5. Move API keys to environment variables (temporary, then secrets manager)

Deliverable: Zero hardcoded secrets, SSL verification enforced in production

---

Day 3: CSRF + CORS Review

Goal: Verify and document authentication strategy

Tasks: 1. Audit all API endpoints for auth method 2. Verify Bearer token usage across all clients 3. Document CSRF strategy decision 4. Review CORS configuration 5. Separate cookie-based web flows if needed

Deliverable: CSRF strategy document, verified safe or fixed

Day 4: Raw SQL Audit

Goal: Ensure all SQL queries are parameterized

Tasks: 1. Find all `cursor.execute` calls 2. Verify parameterization 3. Replace with ORM where possible 4. Add SQL injection tests 5. Document any remaining raw SQL with justification

Deliverable: All SQL queries parameterized, tests added

Day 5: Dependency & Container Scanning

Goal: Establish vulnerability management

Tasks: 1. Run dependency scan (Dependabot/Snyk) 2. Run container scan 3. Document patch SLAs (Critical: 24h, High: 7d, Medium: 30d) 4. Set up automated scanning in CI 5. Create vulnerability tracking spreadsheet

Deliverable: Vulnerability & Patch Program document

Day 6-7: Incident Response + Data Flow

Goal: Complete security documentation

Tasks: 1. Draft Incident Response Plan 2. Create Data Flow Diagram 3. Run tabletop exercise (internal) 4. Document findings 5. Update security questionnaire answers

Deliverables: - Incident Response Plan (2-3 pages) - Data Flow Diagram (1 page) - Tabletop exercise notes

---

🔧 CODE FIXES NEEDED

Pre-Commit Hook (Add to `.git/hooks/pre-commit`)

```
 #!/bin/bash
# Block commits with secrets

SECRET_PATTERNS=(
    "test123"
    "password\s*=\s*['\"]"
    "API_KEY\s*=\s*['\"]"
    "SECRET\s*=\s*['\"]"
    "CERT_NONE"
    "verify\s*=\s*False"
)

for pattern in "${SECRET_PATTERNS[@]}"; do
    if git diff --cached | grep -E "$pattern"; then
        echo "❌ BLOCKED: Potential secret detected: $pattern"
        echo "Remove secrets before committing"
        exit 1
    fi
done
```

Secrets Manager Helper (`deployment_package/backend/core/secrets_manager.py`)

```
"""
AWS Secrets Manager Integration
Loads secrets at startup, caches in memory
"""
import os
import json
import logging
from typing import Optional

logger = logging.getLogger(__name__)

_secrets_cache = {}

def get_secret(secret_name: str, use_cache: bool = True) -> Optional[str]:
    """
    Get secret from AWS Secrets Manager or environment variable

    Priority:
    1. Environment variable (for local dev)
    2. Secrets Manager (for production)
    3. Cache (if enabled)
    """
    # Check environment first (local dev)
    env_key = secret_name.upper().replace('-', '_')
    env_value = os.getenv(env_key)
    if env_value:
        return env_value

    # Check cache
    if use_cache and secret_name in _secrets_cache:
        return _secrets_cache[secret_name]

    # Load from Secrets Manager (production)
    if os.getenv('ENVIRONMENT') == 'production':
        try:
            import boto3
            client = boto3.client('secretsmanager', region_name=os.getenv('AWS_REGIO
            response = client.get_secret_value(SecretId=secret_name)
            secret = json.loads(response['SecretString'])

            # Cache it
            if use_cache:
                _secrets_cache[secret_name] = secret
            return secret
        except Exception as e:
            logger.error(f"Failed to load secret {secret_name}: {e}")
            return None

    return None
```

SSL Verification Guard

```
# Add to deployment_package/backend/core/security_utils.py

import os
import ssl

def get_ssl_context():
    """Get SSL context with production guard"""
    verify_ssl = os.getenv('SSL_VERIFY', 'true').lower() == 'true'
    environment = os.getenv('ENVIRONMENT', 'development')

    # Hard block in production
    if environment == 'production' and not verify_ssl:
        raise ValueError(
            "SSL verification cannot be disabled in production. "
            "Set SSL_VERIFY=true or remove SSL_VERIFY from environment."
        )

    if verify_ssl:
        return ssl.create_default_context()
    else:
        # Only allow in development
        context = ssl.create_default_context()
        context.check_hostname = False
        context.verify_mode = ssl.CERT_NONE
        logger.warning("⚠️ SSL verification disabled (development only)")
        return context
```

📋 VERIFICATION CHECKLIST

Before production launch, verify:

- [ ] Zero hardcoded secrets (run secret scan)
- [ ] SSL verification enforced in production
- [ ] CSRF strategy documented and verified
- [ ] All API keys in Secrets Manager
- [ ] All SQL queries parameterized
- [ ] Dependency scanning enabled
- [ ] Incident Response Plan complete
- [ ] Data Flow Diagram created
- [ ] Security headers configured correctly
- [ ] Rate limiting tested
- [ ] Encryption verified (tokens, database)

🎯 SUCCESS CRITERIA

Production-Ready When: 1. ✅ Secret scan passes (zero hardcoded secrets) 2. ✅ SSL verification cannot be disabled in production 3. ✅ CSRF strategy documented and verified safe 4. ✅ All secrets in Secrets Manager 5. ✅ All SQL queries parameterized 6. ✅ Vulnerability scanning automated 7. ✅ Incident Response Plan documented 8. ✅ Data Flow Diagram complete

Then: Security rating jumps to **9.5/10** - Enterprise-ready