

# Project: CryptoCore - Technical Requirements Document (Sprint 7)

**Sprint Goal:** Implement functions for securely deriving keys from passwords and other keys.

## 1. Project Structure & Repository Hygiene

The codebase must be extended to support key derivation functionality while maintaining existing structure.

ID	Requirement Description	Priority
STR-1	All requirements from previous Sprints (STR-1 to STR-4) <b>must</b> still be met.	Must
STR-2	New source files for key derivation <b>must</b> be created in a logical directory structure.  - <b>Suggested Path:</b> <code>src/kdf/</code> containing <code>pbkdf2.py/pbkdf2.c</code> and <code>hkdf.py/hkdf.c</code>	Must
STR-3	The <code>README.md</code> file <b>must</b> be updated to include: <ul style="list-style-type: none"><li>- Documentation for the new <code>derive</code> command and its options.</li><li>- Explanation of key derivation concepts (key stretching, salting, iteration count).</li><li>- Examples of deriving keys from passwords and creating key hierarchies.</li><li>- Security considerations (minimum iteration counts, salt requirements).</li></ul>	Must
STR-4	The build system <b>must</b> be updated to include any new source files for KDF implementations.	Must

## 2. Command-Line Interface (CLI) Parser

A new subcommand must be implemented for key derivation operations.

ID	Requirement Description	Priority
CLI-1	The tool <b>must</b> support a new subcommand <code>derive</code> for key derivation.	Must
CLI-2	The <code>derive</code> subcommand <b>must</b> accept the following arguments: <ul style="list-style-type: none"><li><code>--password PASSWORD</code> : <b>Must</b> accept a password string. If containing special characters, it should be properly quoted by the shell.</li><li><code>--salt SALT</code> : <b>Must</b> accept a salt as a hexadecimal string. If not provided, the tool <b>should</b> generate a random 16-byte salt.</li><li><code>--iterations COUNT</code> : <b>Must</b> accept an integer iteration count (default: 100,000).</li><li><code>--length LENGTH</code> : <b>Must</b> accept the desired key length in bytes (default: 32).</li><li><code>--algorithm ALGORITHM</code> : <b>Must</b> accept the KDF algorithm (initially only <code>pbkdf2</code>).</li></ul>	Must
CLI-3	The output <b>must</b> be printed to <code>stdout</code> as: <code>KEY_HEX SALT_HEX</code> (both as hexadecimal strings).	Must
CLI-4	The tool <b>should</b> support an optional <code>--output FILE</code> flag to write the derived key to a file.	Should

ID	Requirement Description	Priority
CLI-5	The tool <b>should</b> support reading the password from a file or environment variable for security.	Could

## Example Invocations:

```
# Basic key derivation with specified salt
$ cryptocore derive --password "MySecurePassword123!" --salt a1b2c3d4e5f6012345678901234
> 5f4dcc3b5aa765d61d8327deb882cf992b95990a9151374abd7fe5c8f9b8a8c7 a1b2c3d4e5f601234567

# Key derivation with auto-generated salt
$ cryptocore derive --password "AnotherPassword" --iterations 500000 --length 16
> 8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92 e3b0c44298fc1c149afb

# Derive key and save to file
$ cryptocore derive --password "app_key" --salt fixedappsalt --iterations 10000 --length
```

## 3. Key Derivation Function Implementation

Two key derivation functions must be implemented: PBKDF2 from scratch and a key hierarchy function.

ID	Requirement Description	Priority
KDF-1	<b>PBKDF2-HMAC-SHA256 must</b> be implemented from scratch by the student. <ul style="list-style-type: none"> <li>- The implementation <b>must</b> follow RFC 2898 specification.</li> <li>- It <b>must</b> use the HMAC-SHA256 implementation from Sprint 5.</li> <li>- It <b>must</b> support iteration counts up to at least 1,000,000 efficiently.</li> </ul>	Must
KDF-2	The PBKDF2 implementation <b>must</b> correctly compute: $DK = T1 \parallel T2 \parallel \dots \parallel Tdklen \text{ where } Ti = F(P, S, c, i) \text{ and } F \text{ is the PRF (HMAC-SHA256)}$ $F(P, S, c, i) = U1 \oplus U2 \oplus \dots \oplus Uc \text{ where } U1 = \text{PRF}(P, S \parallel \text{INT\_32\_BE}(i))$ $Uj = \text{PRF}(P, Uj-1)$	Must
KDF-3	The implementation <b>must</b> handle arbitrary password lengths and salt lengths.	Must
KDF-4	The implementation <b>must</b> produce output of the exact length requested by the user.	Must
KDF-5	<b>Key hierarchy function must</b> be implemented for deriving multiple keys from a master key. <ul style="list-style-type: none"> <li>- Function signature: <code>derive_key(master_key: bytes, context: str, length: int) -&gt; bytes</code></li> <li>- The implementation <b>must</b> use HMAC in a deterministic way: <code>HMAC(master_key, context    counter)</code></li> <li>- It <b>must</b> support deriving keys of arbitrary length (by generating enough bytes and truncating).</li> </ul>	Must

## ID      Requirement Description

- The context string **must** be a unique identifier for the key's purpose (e.g., "encryption", "authentication").

**Expected PBKDF2 Implementation Structure:**

```
# Python: src/kdf/pbkdf2.py
import hashlib
import hmac

def pbkdf2_hmac_sha256(password, salt, iterations, dklen):
    """PBKDF2-HMAC-SHA256 implementation"""
    # Password and salt should be bytes
    if isinstance(password, str):
        password = password.encode('utf-8')
    if isinstance(salt, str):
        salt = bytes.fromhex(salt) if all(c in '0123456789abcdefABCDEF' for c in salt) else bytes.fromhex(salt)
    # Calculate number of blocks needed
    blocks_needed = (dklen + 31) // 32 # SHA-256 produces 32-byte output
    derived_key = b''

    for i in range(1, blocks_needed + 1):
        # Compute U1 = HMAC(password, salt || INT_32_BE(i))
        block = hmac_sha256(password, salt + i.to_bytes(4, 'big'))
        u_prev = block

        # Compute U2 through Uc
        for j in range(2, iterations + 1):
            u_curr = hmac_sha256(password, u_prev)
            # XOR u_curr into block
            block = bytes(a ^ b for a, b in zip(block, u_curr))
            u_prev = u_curr

        derived_key += block

    # Return exactly dklen bytes
    return derived_key[:dklen]

def hmac_sha256(key, msg):
    """HMAC-SHA256 using our implementation from Sprint 5"""
    # Use HMAC class from src/mac/hmac.py
    pass
```

**Expected Key Hierarchy Implementation:**

```
# Python: src/kdf/hkdf.py (simplified version for key hierarchy)
def derive_key(master_key, context, length=32):
    """Derive a key from a master key using a deterministic HMAC-based method"""
    if isinstance(context, str):
        context = context.encode('utf-8')

    derived = b''
```

```

counter = 1

while len(derived) < length:
    # T_i = HMAC(master_key, context || counter)
    block = hmac_sha256(master_key, context + counter.to_bytes(4, 'big'))
    derived += block
    counter += 1

return derived[:length]

```

## 4. File I/O for Key Derivation

The key derivation functionality must handle password and salt input securely.

ID	Requirement Description	Priority
IO-1	The password <b>must</b> be read from the command line argument or, if implemented, from a file or environment variable.	Must
IO-2	The salt <b>must</b> be read as a hexadecimal string or generated randomly if not provided.	Must
IO-3	If --output is specified, the derived key <b>must</b> be written to the file as raw binary bytes (not hexadecimal). - The salt should <b>not</b> be written to the key file unless explicitly requested.	Must
IO-4	The tool <b>should</b> clear password from memory as soon as possible after use.	Should

## 5. Testing & Verification

Comprehensive testing must ensure correctness and interoperability with standard implementations.

ID	Requirement Description	Priority
TEST-1	<b>Known-Answer Tests:</b> The PBKDF2 implementation <b>must</b> pass test vectors from RFC 6070. - Specifically, test vectors for PBKDF2 with HMAC-SHA256.	Must
TEST-2	<b>Iteration Test:</b> The implementation <b>must</b> produce the same result when run with the same parameters multiple times.	Must
TEST-3	<b>Length Test:</b> The implementation <b>must</b> correctly derive keys of various lengths (1-100 bytes).	Must
TEST-4	<b>Interoperability Test:</b> The PBKDF2 output <b>must</b> match OpenSSL's implementation:  openssl kdf -keylen 32 -kdfopt pass:PASSWORD -kdfopt salt:SALT -kdfopt iter:ITERATIONS PBKDF2	Must
TEST-5	<b>Key Hierarchy Test:</b> The derive_key function <b>must</b> produce deterministic output for the same inputs.	Must
TEST-6	<b>Context Separation Test:</b> Different context strings <b>must</b> produce completely different keys from the same master key.	Must

ID	Requirement Description	Priority
TEST-7	<b>Salt Randomness Test:</b> When no salt is provided, generated salts <b>must</b> be unique and random (test 1000 generations for no duplicates).	Must
TEST-8	<b>Performance Test:</b> The student <b>should</b> measure and document the time taken for different iteration counts (10k, 100k, 1M).	Should

## Example Test Commands:

```
# Test with RFC 6070 test vector 1
$ cryptocore derive --password "password" --salt 73616c74 --iterations 1 --length 20
> Should produce: 0c60c80f961f0e71f3a9b524af6012062fe037a6

# Test with RFC 6070 test vector 2
$ cryptocore derive --password "password" --salt 73616c74 --iterations 2 --length 20
> Should produce: ea6c014dc72d6f8cccd1ed92ace1d41f0d8de8957

# Interoperability test with OpenSSL
$ cryptocore derive --password "test" --salt 1234567890abcdef --iterations 1000 --length 16
$ openssl kdf -keylen 32 -kdfopt pass:test -kdfopt salt:1234567890abcdef -kdfopt iter:1000
$ diff my_output.txt openssl_output.txt

# Test key hierarchy function
$ python3 -c "
from src.kdf.hkdf import derive_key
master = b'0' * 32
key1 = derive_key(master, 'encryption', 32)
key2 = derive_key(master, 'authentication', 32)
print(f'Key1: {key1.hex()}'')
print(f'Key2: {key2.hex()}'')
print(f'Different: {key1 != key2}'')
"
```

## Example Test Script for RFC 6070 Vectors:

```
# tests/test_pbkdf2_vectors.py
from src.kdf.pbkdf2 import pbkdf2_hmac_sha256

def test_rfc_6070():
    """Test PBKDF2 with RFC 6070 test vectors"""
    test_cases = [
        {
            'password': b'password',
            'salt': b'salt',
            'iterations': 1,
            'dklen': 20,
            'expected': '0c60c80f961f0e71f3a9b524af6012062fe037a6'
        },
        {
            'password': b'password',
            'salt': b'salt',
            'iterations': 2,
            'dklen': 20,
            'expected': 'ea6c014dc72d6f8cccd1ed92ace1d41f0d8de8957'
        }
    ]
```

```
},
{
    'password': b'password',
    'salt': b'salt',
    'iterations': 4096,
    'dklen': 20,
    'expected': '4b007901b765489abead49d926f721d065a429c1'
},
{
    'password': b'passwordPASSWORDpassword',
    'salt': b'saltSALTsaltSALTsaltSALTsaltSALTsalt',
    'iterations': 4096,
    'dklen': 25,
    'expected': '3d2eec4fe41c849b80c8d83662c0e44a8b291a964cf2f07038'
}
]

for i, test in enumerate(test_cases):
    result = pbkdf2_hmac_sha256(
        test['password'],
        test['salt'],
        test['iterations'],
        test['dklen']
    )
    expected_bytes = bytes.fromhex(test['expected'])

    assert result == expected_bytes, f"RFC 6070 test case {i+1} failed"
    print(f"✓ RFC 6070 test case {i+1} passed")

print("All RFC 6070 test vectors passed")
```