OBINexus
11.05.2025
By Nnamdi Michael Okpala

Cryptographic Primitives- A Proposal for a Cryptographic Standard.

Introduction

In modern software systems, cryptographic standards vary greatly depending on use cases, development environments, and security assumptions. The purpose of this proposal is to define a portable, configurable, and clear cryptographic configuration standard—useful for developers building cryptographic systems across languages like Python, Lua, and more.

This proposal introduces a **configuration schema** that defines cryptographic primitives, algorithms, storage, timeouts, and encoding logic. It promotes **best practices** in cryptography by offering a reusable, minimal, and secure default configuration structure.

We emphasise the need for a **standard convention** that all developers can adopt across platforms and languages. This shared standard ensures interoperability, consistency, and long-term maintainability of secure systems. All configurations and components should **support semantic versioning** to track backwards-compatible changes, deprecations, and feature enhancements.

Concept: Cryptographic Primitives

A **cryptographic primitive** refers to a representation of a systematic algorithm that can be reduced into a **primitive** representation, confidentiality, integrity, or authentication. These are usually combined into protocols or systems.

Naming Convention (Proposed)

We propose a naming format for cryptographic components to ensure traceability and version clarity:

This schema is an example of how to encode consistent and secure values across platforms, and how semantic versioning can be embedded directly to support structured upgrades and compatibility tracking.

Goals

Portability: Standard should be easily portable to Python, Lua, C, and other language systems.

Modularity: Each configuration should be component-based and swappable.

Security-Oriented: Focus on best practices (salt, key derivation iterations, strict validation).

Versioned: Every config and primitive must support versioning and validation checks.

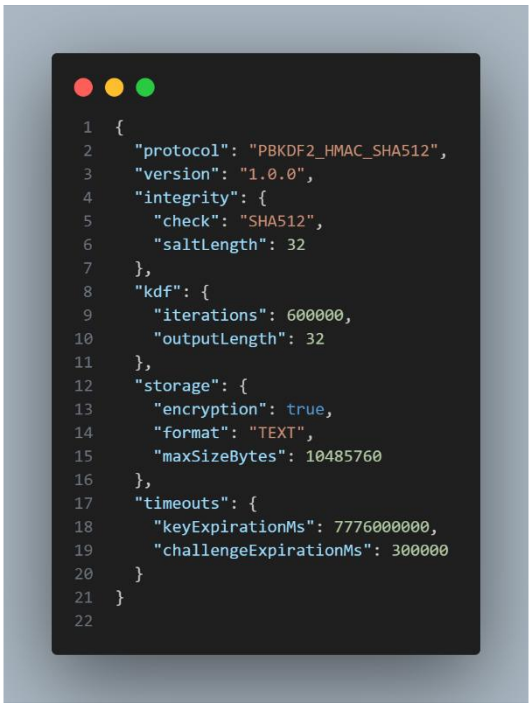
Usable in Realtime and Batch Modes: The config must work for both interactive REPL tools and backend automation.

Semantic Version Support: Every component and schema must include a version field following Semantic Versioning (e.g., MAJOR.MINOR.PATCH).

Best Practices Summary

Always use a strong KDF (e.g., PBKDF2_HMAC_SHA512) with a high iteration count (e.g., 600000).
Prefer SHA512 for hashing, unless space or compatibility demands SHA256.
Enable secure memory and storage encryption for all sensitive data.
Separate timeouts by component type (e.g., key vs. proof).
Validate algorithms explicitly via an **allowedAlgorithms** list.
Ensure semantic versioning is maintained to track and communicate changes to configuration schemas and cryptographic components.

OBINexus
11.05.2025
By Nnamdi Michael Okpala



```json
{
  "protocol": "PBKDF2_HMAC_SHA512",
  "version": "1.0.0",
  "integrity": {
    "check": "SHA512",
    "saltLength": 32
  },
  "kdf": {
    "iterations": 600000,
    "outputLength": 32
  },
  "storage": {
    "encryption": true,
    "format": "TEXT",
    "maxSizeBytes": 10485760
  },
  "timeouts": {
    "keyExpirationMs": 7776000000,
    "challengeExpirationMs": 300000
  }
}
```

Conclusion:

Cryptographic Primitives should be used to enhance strong system integrity, whether in a development system or a real-world application. A strong cryptographic primitive setup will simplify and amplify a cryptography system, given seamless flexibility (modification) of program logic in the future.

On the Right is an exemplification of a JSON file that **stamps** cryptographic protocol primitives.