Ten Build System: Architectural Refactor

1 Architecture Overview

1.1 Core Architectural Principles

Ten is a Nix-inspired build system with two orthogonal dimensions:

1. **Phase**: Represents the stage in the build process
   - `Eval` - Expression evaluation, derivation instantiation
   - `Build` - Output construction, build execution
2. **PrivilegeTier**: Represents the privilege level
   - `Daemon` - Privileged operations (store access, sandbox creation)
   - `Builder` - Unprivileged operations (executing builds)

1.2 Singleton-Based Type Safety

The core architectural transformation implements a singleton-based approach to provide:

- Compile-time enforcement of privilege boundaries
- Runtime evidence for dynamic privilege checks
- Proper phase separation while allowing controlled transitions
- Nix-like security guarantees with clean syntax

1.3 Implementation Requirements

For this universal syntax to work, the following must be implemented:

1. Dependency analysis system to detect true data dependencies
2. Build strategy inference based on derivation inspection
3. Automatic return-derivation detection in builder output
4. Safe phase transitions between evaluation and building
5. Privilege-aware operations that work with the unified syntax
6. Type-safe monadic interface that preserves expressivity

This unified approach maintains Ten's elegant, minimal syntax while leveraging Haskell's type system to handle the complexity internally, giving users a simple interface to a powerful build system.

2 File-by-File Changes

2.1 src/Ten/Core.hs

Ten/Core.hs Changes

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE RankNTypes #-}

module Ten.Core where

import Data.Singletons.TH
import Data.Kind (Type)
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Except

-- Define the core kinds
data Phase = Eval | Build
data PrivilegeTier = Daemon | Builder

-- Generate singletons using Template Haskell
$(genSingletons [''Phase, ''PrivilegeTier])
$(singDecideInstances [''Phase, ''PrivilegeTier])

-- Type families for permissions and capabilities
type family CanAccessStore (t :: PrivilegeTier) :: Bool where
  CanAccessStore 'Daemon = 'True
  CanAccessStore 'Builder = 'False

type family CanCreateSandbox (t :: PrivilegeTier) :: Bool where
  CanCreateSandbox 'Daemon = 'True
  CanCreateSandbox 'Builder = 'False

type family CanEvaluate (p :: Phase) :: Bool where
  CanEvaluate 'Eval = 'True
  CanEvaluate 'Build = 'False

type family CanBuild (p :: Phase) :: Bool where
  CanBuild 'Build = 'True
  CanBuild 'Eval = 'False

-- Error types
data BuildError =
    EvalError Text
  | BuildFailed Text
  | StoreError Text
  | SandboxError Text
  | PrivilegeError Text
  | PhaseError Text
  -- Other error types

-- Environment and state types
data BuildEnv = BuildEnv {
    workDir :: FilePath,
    storeLocation :: FilePath,
    verbosity :: Int,
    -- Runtime representation for privilege checking
```

```
    currentPrivilege :: SomePrivilegeTier,
    currentPhaseRep :: SomePhase,
    -- Other fields
}

data BuildState = BuildState {
    -- State fields
}

-- Core monad definition with singleton evidence
newtype TenM (p :: Phase) (t :: PrivilegeTier) a = TenM {
    runTenM :: SPhase p -> SPrivilegeTier t ->
               ReaderT BuildEnv (StateT BuildState (ExceptT BuildError IO)) a
}

-- Instances
instance Functor (TenM p t) where
    fmap f (TenM g) = TenM $ \sp st -> fmap f (g sp st)

instance Applicative (TenM p t) where
    pure a = TenM $ \_ _ -> pure a
    (TenM f) <*> (TenM g) = TenM $ \sp st -> f sp st <*> g sp st

instance Monad (TenM p t) where
    (TenM m) >>= f = TenM $ \sp st -> do
        a <- m sp st
        let (TenM m') = f a
        m' sp st

-- Helper for singleton creation
phase :: forall (p :: Phase). SingI p => SPhase p
phase = sing

privilege :: forall (t :: PrivilegeTier). SingI t => SPrivilegeTier t
privilege = sing

-- Phase transition with singleton evidence
transitionPhase ::
    forall (p :: Phase) (q :: Phase) (t :: PrivilegeTier) a.
    (SingI p, SingI q) =>
    TenM p t a -> TenM q t a
transitionPhase (TenM m) = TenM $ \_ st ->
    m (phase @p) st

-- Privilege transition with singleton evidence
-- Can only drop privileges, never gain them
transitionPrivilege ::
    forall (p :: Phase) (t :: PrivilegeTier) (t' :: PrivilegeTier) a.
    (SingI t, SingI t', CanAccessStore t ~ 'True, CanAccessStore t' ~ 'False) =>
    TenM p t a -> TenM p t' a
transitionPrivilege (TenM m) = TenM $ \sp _ ->
    m sp (privilege @t)

-- Runtime wrappers for operations with compile-time and runtime checks
withStore :: forall (p :: Phase) (t :: PrivilegeTier) a.
    (SingI t, CanAccessStore t ~ 'True) =>
    (SPrivilegeTier t -> TenM p t a) -> TenM p t a
withStore f = TenM $ \sp st ->
    let (TenM m) = f st
    in m sp st

withBuild :: forall (p :: Phase) (t :: PrivilegeTier) a.
    (SingI p, CanBuild p ~ 'True) =>
    (SPhase p -> TenM p t a) -> TenM p t a
withBuild f = TenM $ \sp st ->
    let (TenM m) = f sp
    in m sp st
```

```haskell
-- Execution functions
runTen :: forall (p :: Phase) (t :: PrivilegeTier) a.
    (SingI p, SingI t) =>
    TenM p t a -> BuildEnv -> BuildState -> IO (Either BuildError (a, BuildState))
runTen (TenM m) env state =
    runExceptT $ runStateT (runReaderT (m (phase @p) (privilege @t)) env) state

-- Other core functions and instances
```

2.2 src/Ten/Derivation.hs

Ten/Derivation.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE FlexibleContexts #-}

module Ten.Derivation where

import Data.Singletons
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Except

import Ten.Core
import Ten.Store

-- Core derivation creation - works in Eval phase
mkDerivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    DerivationAttrs -> TenM 'Eval t Derivation
mkDerivation attrs = TenM $ \sEval st -> do
    -- Implementation details
    -- ...

-- Instantiate a derivation for building
instantiateDerivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPhase 'Build -> SPrivilegeTier t ->
    Derivation -> TenM 'Build t ()
instantiateDerivation = -- Implementation

-- Store a derivation with proper privilege checking
storeDerivation :: forall (p :: Phase) (t :: PrivilegeTier).
    (SingI t, CanAccessStore t ~ 'True) =>
    SPrivilegeTier t -> Derivation -> TenM p t StorePath
storeDerivation st drv = withStore $ \st' ->
    -- Implementation

-- Retrieve a derivation - available in any context
retrieveDerivation :: forall (p :: Phase) (t :: PrivilegeTier).
    (SingI t) =>
    StorePath -> TenM p t (Maybe Derivation)
retrieveDerivation = -- Implementation

-- Return-continuation pattern (monadic join)
joinDerivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPhase 'Build -> SPrivilegeTier t ->
    Derivation -> TenM 'Build t Derivation
joinDerivation sBuild st derivation = TenM $ \_ _ -> do
    -- Implementation with runtime checks based on singleton evidence
    -- ...
```

```haskell
-- Helper for safely transitioning to build phase
evaluateThenBuild :: forall (t :: PrivilegeTier).
    (SingI t) =>
    TenM 'Eval t Derivation -> TenM 'Build t BuildResult
evaluateThenBuild eval = TenM $ \sBuild st -> do
    -- Implementation with safe phase transition
    -- ...

-- Universal do-notation entry point
derivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    DerivationAttrs -> TenM 'Eval t Derivation
derivation = mkDerivation
```

2.3 src/Ten/DB/Core.hs

Ten/DB/Core.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Ten.DB.Core where

import Data.Singletons
import qualified Database.SQLite.Simple as SQLite
import Database.SQLite.Simple (Connection, Query, ToRow, FromRow, Only)

import Ten.Core

-- Database operations require daemon privileges
withDatabase :: forall (p :: Phase) a.
    (SingI p) =>
    SPrivilegeTier 'Daemon ->
    FilePath -> Int -> (Database -> TenM p 'Daemon a) -> TenM p 'Daemon a
withDatabase st dbPath busyTimeout action = TenM $ \sp _ -> do
    -- Implementation with runtime privilege check
    -- ...

-- Execute with parameters and return affected rows
dbExecute :: forall (p :: Phase) q.
    (ToRow q, SingI p) =>
    SPrivilegeTier 'Daemon -> Database -> Query -> q -> TenM p 'Daemon Int64
dbExecute st db query params = TenM $ \sp _ -> do
    -- Implementation
    -- ...

-- Execute with parameters, discard result
dbExecute_ :: forall (p :: Phase) q.
    (ToRow q, SingI p) =>
    SPrivilegeTier 'Daemon -> Database -> Query -> q -> TenM p 'Daemon ()
dbExecute_ st db query params = void $ dbExecute st db query params

-- Execute without parameters
dbExecuteSimple_ :: forall (p :: Phase).
    (SingI p) =>
    SPrivilegeTier 'Daemon -> Database -> Query -> TenM p 'Daemon ()
dbExecuteSimple_ st db query = TenM $ \sp _ -> do
    -- Implementation
    -- ...

-- Query with parameters
dbQuery :: forall (p :: Phase) q r.
    (ToRow q, FromRow r, SingI p) =>
```

```
    SPrivilegeTier 'Daemon -> Database -> Query -> q -> TenM p 'Daemon [r]
dbQuery st db query params = TenM $ \sp _ -> do
    -- Implementation
    -- ...

-- Query without parameters
dbQuery_ :: forall (p :: Phase) r.
    (FromRow r, SingI p) =>
    SPrivilegeTier 'Daemon -> Database -> Query -> TenM p 'Daemon [r]
dbQuery_ st db query = TenM $ \sp _ -> do
    -- Implementation
    -- ...

-- Higher-level TenM-integrated operations with proper privilege handling

-- Operations that can work in any phase but require daemon privileges
-- ...
```

2.4 src/Ten/Sandbox.hs

Ten/Sandbox.hs Changes

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Ten.Sandbox where

import Data.Singletons
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Except
import Data.Set (Set)
import qualified Data.Set as Set

import Ten.Core
import Ten.Store

-- Sandbox configuration
data SandboxConfig = SandboxConfig {
    -- Configuration fields
}

-- Create a sandbox (daemon privilege required)
createSandbox ::
    SPrivilegeTier 'Daemon ->
    FilePath -> SandboxConfig -> TenM 'Build 'Daemon FilePath
createSandbox st dir config = TenM $ \sp _ -> do
    -- Implementation with privilege checks
    -- ...

-- Use a sandbox from any privilege context
useSandbox :: forall (t :: PrivilegeTier) a.
    (SingI t) =>
    SPrivilegeTier t ->
    FilePath -> TenM 'Build t a -> TenM 'Build t a
useSandbox st path action = TenM $ \sp _ -> do
    -- Implementation
    -- ...

-- Main sandbox entry point with proper dispatching based on context
withSandbox :: forall (t :: PrivilegeTier) a.
    (SingI t) =>
    SPrivilegeTier t ->
    Set StorePath -> SandboxConfig -> (FilePath -> TenM 'Build t a) -> TenM 'Build t a
```

```haskell
withSandbox st inputs config action = TenM $ \sp _ -> do
    env <- ask

    -- Dispatch to appropriate implementation based on singleton evidence
    case (fromSing st) of
        Daemon ->
            -- Use direct sandbox creation for daemon
            let (TenM m) = withSandboxDaemon inputs config action
            in m sp st

        Builder ->
            -- Use protocol-based sandbox for builder
            let (TenM m) = withSandboxViaProtocol inputs config action
            in m sp st

-- Implementation for daemon context
withSandboxDaemon ::
    Set StorePath -> SandboxConfig -> (FilePath -> TenM 'Build 'Daemon a) -> TenM 'Build 'Daemon a
withSandboxDaemon = -- Implementation

-- Implementation for builder context
withSandboxViaProtocol ::
    Set StorePath -> SandboxConfig -> (FilePath -> TenM 'Build 'Builder a) -> TenM 'Build 'Builder a
withSandboxViaProtocol = -- Implementation

-- Get sandbox directory (daemon only)
getSandboxDir ::
    SPrivilegeTier 'Daemon ->
    BuildEnv -> TenM 'Build 'Daemon FilePath
getSandboxDir = -- Implementation

-- Setup sandbox with namespace isolation (daemon only)
setupSandbox ::
    SPrivilegeTier 'Daemon ->
    FilePath -> SandboxConfig -> TenM 'Build 'Daemon ()
setupSandbox = -- Implementation

-- Functions for privilege dropping within sandbox
dropPrivileges ::
    SPrivilegeTier 'Daemon ->
    String -> String -> TenM 'Build 'Daemon ()
dropPrivileges = -- Implementation

-- Setup environment for builder
prepareSandboxEnvironment ::
    BuildEnv -> BuildState -> FilePath -> Map Text Text -> Map Text Text
prepareSandboxEnvironment = -- Implementation
```

2.5 src/Ten/Build.hs

Ten/Build.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Ten.Build where

import Data.Singletons
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Except
import Data.Set (Set)
import qualified Data.Set as Set
```

```haskell
import Ten.Core
import Ten.Store
import Ten.Sandbox
import Ten.Derivation

-- Build result type
data BuildResult = BuildResult {
    -- Result fields
}

-- Universal entry point with privilege dispatching
buildDerivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> Derivation -> TenM 'Build t BuildResult
buildDerivation st derivation = TenM $ \sp _ -> do
    -- Dispatch based on privilege context
    case (fromSing st) of
        Daemon ->
            let (TenM m) = buildDerivationDaemon st derivation
            in m sp st

        Builder ->
            let (TenM m) = buildDerivationBuilder st derivation
            in m sp st

-- Daemon-specific implementation
buildDerivationDaemon ::
    SPrivilegeTier 'Daemon -> Derivation -> TenM 'Build 'Daemon BuildResult
buildDerivationDaemon st derivation = TenM $ \sp _ -> do
    -- Implementation with full privileges
    -- ...

-- Builder-specific implementation
buildDerivationBuilder ::
    SPrivilegeTier 'Builder -> Derivation -> TenM 'Build 'Builder BuildResult
buildDerivationBuilder st derivation = TenM $ \sp _ -> do
    -- Implementation with limited privileges
    -- ...

-- Build strategy selection
buildApplicativeStrategy :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> Derivation -> TenM 'Build t BuildResult
buildApplicativeStrategy = -- Implementation

buildMonadicStrategy :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> Derivation -> TenM 'Build t BuildResult
buildMonadicStrategy = -- Implementation

-- Build result handling
collectBuildResult :: forall (t :: PrivilegeTier).
    (SingI t, CanAccessStore t ~ 'True) =>
    SPrivilegeTier t -> Derivation -> FilePath -> TenM 'Build t (Set StorePath)
collectBuildResult = -- Implementation

verifyBuildResult :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> Derivation -> BuildResult -> TenM 'Build t Bool
verifyBuildResult = -- Implementation

-- Return-continuation handling
checkForReturnedDerivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> FilePath -> TenM 'Build t (Maybe Derivation)
checkForReturnedDerivation = -- Implementation
```

```haskell
handleReturnedDerivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> BuildResult -> TenM 'Build t Derivation
handleReturnedDerivation = -- Implementation

-- Dependency handling for parallel builds
buildDependenciesConcurrently :: forall (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> [Derivation] -> TenM 'Build t (Map String (Either BuildError BuildResult))
buildDependenciesConcurrently = -- Implementation

-- Resource management
runBuilder ::
    SPrivilegeTier 'Daemon ->
    BuilderEnv -> TenM 'Build 'Daemon (Either Text (ExitCode, String, String))
runBuilder = -- Implementation
```

2.6 src/Ten/Store.hs

Ten/Store.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Ten.Store where

import Data.Singletons
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Except
import qualified Data.ByteString as BS
import Data.Text (Text)
import qualified Data.Text as T

import Ten.Core

-- Store path operations that require daemon privileges
addToStore ::
    SPrivilegeTier 'Daemon ->
    Text -> BS.ByteString -> TenM p 'Daemon StorePath
addToStore st nameHint content = TenM $ \sp _ -> do
    -- Implementation with privilege check
    -- ...

storeFile ::
    SPrivilegeTier 'Daemon ->
    FilePath -> TenM p 'Daemon StorePath
storeFile st path = TenM $ \sp _ -> do
    -- Implementation with privilege check
    -- ...

storeDirectory ::
    SPrivilegeTier 'Daemon ->
    FilePath -> TenM p 'Daemon StorePath
storeDirectory st path = TenM $ \sp _ -> do
    -- Implementation with privilege check
    -- ...

removeFromStore ::
    SPrivilegeTier 'Daemon ->
    StorePath -> TenM p 'Daemon ()
removeFromStore st path = TenM $ \sp _ -> do
    -- Implementation with privilege check
    -- ...
```

```haskell
-- Operations available in any context
storePathExists :: forall (p :: Phase) (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> StorePath -> TenM p t Bool
storePathExists st path = TenM $ \sp _ -> do
    -- Context-aware implementation
    -- ...

readFromStore :: forall (p :: Phase) (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> StorePath -> TenM p t BS.ByteString
readFromStore st path = TenM $ \sp _ -> do
    -- Context-aware implementation
    -- ...

verifyStorePath :: forall (p :: Phase) (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> StorePath -> TenM p t Bool
verifyStorePath st path = TenM $ \sp _ -> do
    -- Context-aware implementation
    -- ...

-- Protocol-based operations for builder context
requestAddToStore ::
    SPrivilegeTier 'Builder ->
    Text -> BS.ByteString -> TenM p 'Builder StorePath
requestAddToStore st nameHint content = TenM $ \sp _ -> do
    -- Implementation using protocol
    -- ...

requestReadFromStore ::
    SPrivilegeTier 'Builder ->
    StorePath -> TenM p 'Builder BS.ByteString
requestReadFromStore st path = TenM $ \sp _ -> do
    -- Implementation using protocol
    -- ...

-- Helper for GC lock path (fix ambiguity)
getGCLockPath :: BuildEnv -> FilePath
getGCLockPath env = storeLocation env </> "var/ten/gc.lock"
```

2.7 src/Ten/GC.hs

Ten/GC.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Ten.GC where

import Data.Singletons
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Except
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T

import Ten.Core
import Ten.Store

-- GC operations require daemon privileges
```

```haskell
collectGarbage ::
    SPrivilegeTier 'Daemon ->
    TenM p 'Daemon GCStats
collectGarbage st = TenM $ \sp _ -> do
    -- Implementation with privilege check
    -- ...

-- Root management (daemon only)
addRoot ::
    SPrivilegeTier 'Daemon ->
    StorePath -> Text -> Bool -> TenM p 'Daemon GCRoot
addRoot st path name permanent = TenM $ \sp _ -> do
    -- Implementation with privilege check
    -- ...

removeRoot ::
    SPrivilegeTier 'Daemon ->
    GCRoot -> TenM p 'Daemon ()
removeRoot st root = TenM $ \sp _ -> do
    -- Implementation with privilege check
    -- ...

-- Operations that can be used from either context
isReachable :: forall (p :: Phase) (t :: PrivilegeTier).
    (SingI t) =>
    SPrivilegeTier t -> StorePath -> TenM p t Bool
isReachable st path = TenM $ \sp _ -> do
    -- Context-aware implementation
    -- ...

-- Protocol-based operations for builder context
requestGarbageCollection ::
    SPrivilegeTier 'Builder ->
    TenM p 'Builder GCStats
requestGarbageCollection st = TenM $ \sp _ -> do
    -- Implementation using protocol
    -- ...

requestAddRoot ::
    SPrivilegeTier 'Builder ->
    StorePath -> Text -> Bool -> TenM p 'Builder GCRoot
requestAddRoot st path name permanent = TenM $ \sp _ -> do
    -- Implementation using protocol
    -- ...
```

2.8 src/Ten/Graph.hs

Ten/Graph.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Ten.Graph where

import Data.Singletons
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Except
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import Data.Text (Text)
import qualified Data.Text as T
```

```haskell
import Ten.Core
import Ten.Store
import Ten.Derivation

-- Replace PrivCtxConstraint with PrivTierConstraint
class PrivTierConstraint (t :: PrivilegeTier)
instance PrivTierConstraint 'Daemon
instance PrivTierConstraint 'Builder

-- Graph operations with proper constraints
createBuildGraph :: forall (t :: PrivilegeTier).
    (PrivTierConstraint t, SingI t) =>
    SPrivilegeTier t -> Set StorePath -> Set Derivation -> TenM 'Eval t BuildGraph
createBuildGraph st requestedOutputs derivations = TenM $ \sp _ -> do
    -- Implementation
    -- ...

validateGraph :: forall (t :: PrivilegeTier).
    (PrivTierConstraint t, SingI t) =>
    SPrivilegeTier t -> BuildGraph -> TenM 'Eval t GraphProof
validateGraph st graph = TenM $ \sp _ -> do
    -- Implementation
    -- ...

detectCycles :: forall (t :: PrivilegeTier).
    (PrivTierConstraint t, SingI t) =>
    SPrivilegeTier t -> BuildGraph -> TenM 'Eval t Bool
detectCycles st graph = TenM $ \sp _ -> do
    -- Implementation
    -- ...

topologicalSort :: forall (t :: PrivilegeTier).
    (PrivTierConstraint t, SingI t) =>
    SPrivilegeTier t -> BuildGraph -> TenM 'Eval t [BuildNode]
topologicalSort st graph = TenM $ \sp _ -> do
    -- Implementation
    -- ...

detectRecursionCycle :: forall (p :: Phase) (t :: PrivilegeTier).
    (PrivTierConstraint t, SingI t, SingI p) =>
    SPhase p -> SPrivilegeTier t -> [Derivation] -> TenM p t Bool
detectRecursionCycle sp st derivations = TenM $ \_ _ -> do
    -- Implementation
    -- ...

-- Other graph operations with proper singletons
```

2.9 src/Ten/CLI.hs

Ten/CLI.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE GADTs #-}

module Ten.CLI (
    -- Command types
    Command(..),
    Options(..),
    defaultOptions,

    -- Command parsing
```

```
    parseArgs,

    -- Context-aware command execution
    runCommand,
    executeInContext,

    -- Privilege-aware dispatch
    dispatchCommand,
    elevatePrivileges,

    -- Context handlers
    handleBuild,
    handleEval,
    handleStore,
    handleGC,

    -- Daemon communication
    withDaemonConnection,
    requestFromDaemon,

    -- Utility functions
    isPrivilegedOperation,
    ensureDirectory,

    -- Error reporting
    reportError,
    reportPrivilegeError
) where

import Control.Monad
import Control.Monad.Reader (ask, asks)
import Control.Monad.State (get, modify)
import Control.Monad.Except (throwError, catchError)
import Control.Monad.IO.Class (liftIO)
import Data.Text (Text)
import qualified Data.Text as T
import System.Environment (getArgs, getProgName)
import System.Exit (exitSuccess, exitFailure)
import System.Directory (doesFileExist, createDirectoryIfMissing)
import System.FilePath ((</>))
import System.Posix.User (getEffectiveUserID)
import qualified System.Process as Process

import Ten.Core
import Ten.Store
import Ten.Build
import Ten.Derivation
import Ten.Graph
import Ten.Daemon.Protocol
import Ten.Daemon.Client

-- | Run a command with context-aware privilege handling
runCommand :: Command -> Options -> IO ()
runCommand cmd options = do
    -- Determine required privilege tier for this command
    let requiredPrivilege = commandPrivilege cmd

    -- Check current privileges
    currentUID <- getEffectiveUserID
    let hasPrivilege = currentUID == 0

    -- Set up environment based on privilege context
    env <- if hasPrivilege
           then setupDaemonEnv options
           else setupBuilderEnv options

    -- Execute command based on privilege context
```

```
    result <- case (requiredPrivilege, hasPrivilege) of
        (DaemonRequired, True) ->
            -- Execute directly with daemon privileges
            runTenDaemon (executeCommand cmd) env (initBuildState (commandPhase cmd))

        (DaemonRequired, False) ->
            -- Need to use daemon protocol
            withDaemonConnection options $ \conn ->
                runTenBuilder (requestCommand cmd conn) env (initBuildState (commandPhase cmd))

        (BuilderSufficient, _) ->
            -- Can run in either context
            if hasPrivilege
                then runTenDaemon (executeCommand cmd) env (initBuildState (commandPhase cmd))
                else runTenBuilder (executeCommand cmd) env (initBuildState (commandPhase cmd))

    -- Handle result
    case result of
        Left err -> reportError err >> exitFailure
        Right _ -> exitSuccess

-- | Determine which privilege tier is needed for a command
commandPrivilege :: Command -> PrivilegeRequirement
commandPrivilege = \case
    Build _ -> BuilderSufficient       -- Build can use either context
    Eval _ -> BuilderSufficient        -- Eval can use either context
    Store (StoreAdd _) -> DaemonRequired    -- Adding to store needs daemon privileges
    Store (StoreList) -> BuilderSufficient  -- Listing can use either
    Store (StoreGC) -> DaemonRequired       -- GC needs daemon privileges
    GC _ -> DaemonRequired                  -- GC needs daemon privileges
    Derivation (RegisterDerivation _) -> DaemonRequired  -- Registration needs daemon
    Derivation (QueryDerivation _) -> BuilderSufficient  -- Query works in either context
    Help -> BuilderSufficient              -- Help works in any context
    Version -> BuilderSufficient           -- Version info works in any context

-- | Determine which phase a command operates in
commandPhase :: Command -> Phase
commandPhase = \case
    Build _ -> Build              -- Build command uses Build phase
    Eval _ -> Eval                -- Eval command uses Eval phase
    Store _ -> Build              -- Store commands use Build phase
    GC _ -> Build                 -- GC uses Build phase
    Derivation _ -> Eval          -- Derivation commands use Eval phase
    Help -> Build                 -- Help works in any phase
    Version -> Build              -- Version works in any phase

-- | Set up the daemon environment (privileged)
setupDaemonEnv :: Options -> IO BuildEnv
setupDaemonEnv options = do
    -- Create store and work directories
    storePath <- resolveStorePath options
    workDir <- resolveWorkDir options
    createDirectoryIfMissing True storePath
    createDirectoryIfMissing True workDir

    return $ initDaemonEnv workDir storePath (Just "root")

-- | Set up the builder environment (unprivileged)
setupBuilderEnv :: Options -> IO BuildEnv
setupBuilderEnv options = do
    -- Create store and work directories
    storePath <- resolveStorePath options
    workDir <- resolveWorkDir options
    createDirectoryIfMissing True workDir

    -- Establish daemon connection if needed and available
    mConn <- if optUseDaemon options
```

```
            then tryConnectDaemon options
            else return Nothing

    case mConn of
        Just conn -> return $ initClientEnv workDir storePath conn
        Nothing -> return $ initBuildEnv workDir storePath

-- | Try to connect to the daemon
tryConnectDaemon :: Options -> IO (Maybe DaemonConnection)
tryConnectDaemon options = do
    -- Get socket path from options or default
    socketPath <- case optDaemonSocket options of
        Just path -> return path
        Nothing -> getDefaultSocketPath

    -- Try to connect
    runExceptT $ do
        credentials <- getUserCredentials
        connectToDaemon socketPath credentials

-- | Execute commands in the appropriate phase and privilege context
executeCommand :: Command -> TenM (PhaseOf cmd) (ContextOf cmd) ()
executeCommand = \case
    -- Eval phase commands
    Eval file ->
        evalFile file

    -- Build phase commands
    Build file ->
        buildFile file

    Store cmd ->
        executeStoreCommand cmd

    GC force ->
        executeGCCommand force

    -- Help and version work in any phase/context
    Help ->
        showHelp

    Version ->
        showVersion

-- | Execute a request via the daemon
requestCommand :: Command -> DaemonConnection -> TenM (PhaseOf cmd) 'Builder ()
requestCommand cmd conn = do
    -- Create appropriate request based on command
    request <- createRequestForCommand cmd

    -- Send request to daemon
    response <- sendToDaemon conn request

    -- Handle response
    handleDaemonResponse response

-- | Create a daemon request for a command
createRequestForCommand :: Command -> TenM p 'Builder DaemonRequest
createRequestForCommand = \case
    Build file -> do
        -- Check file existence first
        exists <- liftIO $ doesFileExist file
        unless exists $
            throwError $ InputNotFound file

        return $ BuildFileRequest file
```

```haskell
    Store (StoreAdd file) -> do
        -- Read file content first
        content <- liftIO $ BS.readFile file
        return $ StoreContentRequest (T.pack $ takeFileName file) content

    GC force ->
        return $ GCRequest force

    cmd ->
        throwError $ ProtocolError $ "Command not supported via protocol: " <> T.pack (show cmd)

-- | Handle a daemon response
handleDaemonResponse :: DaemonResponse -> TenM p 'Builder ()
handleDaemonResponse = \case
    BuildComplete result ->
        liftIO $ showBuildResult result

    StorePathResponse path ->
        liftIO $ putStrLn $ "Stored at: " ++ T.unpack (storePathToText path)

    GCComplete stats ->
        liftIO $ showGCStats stats

    ErrorResponse err ->
        throwError err

    resp ->
        throwError $ ProtocolError $ "Unexpected daemon response: " <> T.pack (show resp)

-- | Helper to execute store commands with appropriate context dispatching
executeStoreCommand :: StoreCommand -> TenM 'Build ctx ()
executeStoreCommand cmd = do
    ctx <- asks privilegeContext
    case (cmd, ctx) of
        -- Commands that require daemon privileges
        (StoreAdd file, 'Daemon) ->
            addFileToStore file

        (StoreAdd _, 'Builder) ->
            throwError $ PrivilegeError "Adding to store requires daemon privileges"

        (StoreGC, 'Daemon) ->
            void collectGarbage

        (StoreGC, 'Builder) ->
            throwError $ PrivilegeError "Garbage collection requires daemon privileges"

        -- Commands that work in either context
        (StoreList, _) ->
            listStoreContents

        (StoreVerify path, _) ->
            verifyAndShowStorePath path

-- | Execute GC with privilege checking
executeGCCommand :: Bool -> TenM 'Build ctx ()
executeGCCommand force = do
    ctx <- asks privilegeContext
    case ctx of
        'Daemon -> do
            -- Can run directly in daemon context
            stats <- withGCLock collectGarbage
            liftIO $ showGCStats stats

        'Builder ->
            -- Need daemon in builder context
            throwError $ PrivilegeError "Garbage collection requires daemon privileges"
```

```haskell
-- | Display help text
showHelp :: TenM p ctx ()
showHelp = liftIO $ do
    putStrLn "Ten - A pure functional build system"
    putStrLn "Usage: ten COMMAND [OPTIONS]"
    -- Help text continues...

-- | Display version information
showVersion :: TenM p ctx ()
showVersion = liftIO $ do
    putStrLn "Ten version 0.1.0"
    putStrLn "Copyright (C) 2025"

-- | Privilege requirement for commands
data PrivilegeRequirement = DaemonRequired | BuilderSufficient
    deriving (Show, Eq)

-- | Type families for mapping commands to appropriate phase and context
type family PhaseOf (cmd :: Command) :: Phase
type family ContextOf (cmd :: Command) :: PrivilegeTier
```

2.10 src/Ten/Daemon/Protocol.hs

Ten/Daemon/Protocol.hs Changes

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE RankNTypes #-}

module Ten.Daemon.Protocol (
    -- Protocol versions
    ProtocolVersion(..),
    currentProtocolVersion,
    compatibleVersions,

    -- Phase and Privilege-aware message types
    Message(..),
    PrivilegedMessage(..),
    UnprivilegedMessage(..),
    RequestTag(..),
    ResponseTag(..),

    -- Capability-checked request/response
    RequestMessage(..),
    ResponseMessage(..),
    DaemonCapability(..),

    -- Privilege-tagged request/response
    DaemonRequest(..),
    DaemonResponse(..),
    RequestPrivilege(..),

    -- Authentication types with privilege evidence
    AuthRequest(..),
    AuthResult(..),

    -- Verification and capability checking
    verifyCapabilities,
    checkPrivilegeRequirement,
```

```
    PrivilegeRequirement(..),

    -- Build tracking with privilege awareness
    BuildRequestInfo(..),
    BuildStatusUpdate(..),
    defaultBuildRequestInfo,

    -- Serialization functions with privilege checks
    serializeMessage,
    deserializeMessage,
    serializePrivilegedRequest,
    serializeUnprivilegedRequest,
    deserializeResponse,

    -- Protocol framing
    createRequestFrame,
    parseRequestFrame,
    createResponseFrame,
    parseResponseFrame,

    -- Socket communication with privilege boundaries
    sendRequest,
    receiveResponse,
    sendResponse,
    receiveRequest,

    -- Connection management
    ProtocolHandle(..),
    createHandle,
    closeHandle,
    withProtocolHandle,

    -- Exception types
    ProtocolError(..),
    PrivilegeError(..)
) where

import Control.Concurrent (forkIO, killThread, threadDelay, myThreadId)
import Control.Concurrent.MVar
import Control.Exception (Exception, throwIO, bracket, try, SomeException)
import Control.Monad (unless, when, foldM)
import Data.Aeson ((.:), (.=))
import qualified Data.Aeson as Aeson
import qualified Data.Aeson.Types as Aeson
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy as LBS
import qualified Data.ByteString.Builder as Builder
import Data.List (intercalate)
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import Data.Maybe (fromMaybe, isNothing, isJust, catMaybes)
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import qualified Data.Text.IO as TIO
import Data.Time.Clock (UTCTime, getCurrentTime, diffUTCTime)
import Data.Word (Word32, Word64)
import GHC.Generics (Generic)
import Network.Socket (Socket, close)
import qualified Network.Socket.ByteString as NByte
import System.Exit (ExitCode(ExitSuccess, ExitFailure))
import System.IO (Handle, IOMode(..), withFile, hClose, hFlush)
import System.IO.Error (isEOFError)
import Text.Read (readMaybe)
```

```haskell
-- Import Ten modules (with singleton types)
import Ten.Core

-- | Protocol version
data ProtocolVersion = ProtocolVersion {
    protocolMajor :: Int,
    protocolMinor :: Int,
    protocolPatch :: Int
} deriving (Eq, Generic)

-- | Daemon capabilities - tied to privilege tiers
data DaemonCapability =
    StoreAccess             -- requires CanAccessStore t ~ 'True
  | SandboxCreation         -- requires CanCreateSandbox t ~ 'True
  | GarbageCollection       -- requires CanAccessStore t ~ 'True
  | DerivationRegistration  -- requires CanAccessStore t ~ 'True
  | DerivationBuild         -- available to all
  | StoreQuery              -- available to all
  | BuildQuery              -- available to all
  deriving (Show, Eq, Ord, Bounded, Enum)

-- | Privilege requirement for operations
data PrivilegeRequirement =
    DaemonRequired    -- Requires daemon privileges
  | BuilderSufficient -- Can be done from builder context
  deriving (Show, Eq)

-- | Request privilege tagging
data RequestPrivilege =
    PrivilegedRequest   -- Must be run in daemon context
  | UnprivilegedRequest -- Can be run in either context
  deriving (Show, Eq)

-- | Phase and privilege-aware message type
data Message (p :: Phase) (t :: PrivilegeTier) where
    -- Messages that require daemon privileges
    PrivilegedMsg :: CanAccessStore t ~ 'True =>
                     RequestMessage -> Message p t

    -- Messages that can be sent from any context
    UnprivilegedMsg :: RequestMessage -> Message p t

-- | Phase and privilege-aware response type
data Response (p :: Phase) (t :: PrivilegeTier) where
    -- Responses that include privileged data
    PrivilegedResp :: CanAccessStore t ~ 'True =>
                      ResponseMessage -> Response p t

    -- Responses that can be received in any context
    UnprivilegedResp :: ResponseMessage -> Response p t

    -- Error responses are always available
    ErrorResp :: BuildError -> Response p t

-- | Protocol errors
data ProtocolError
    = ProtocolParseError Text
    | VersionMismatch ProtocolVersion ProtocolVersion
    | MessageTooLarge Word32
    | ConnectionClosed
    | AuthenticationFailed Text
    | OperationFailed Text
    | InvalidRequest Text
    | InternalError Text
    | PrivilegeViolation Text  -- Added for privilege violations
    deriving (Show, Eq)
```

```haskell
instance Exception ProtocolError

-- | Privilege errors
data PrivilegeError
    = InsufficientPrivileges Text
    | PrivilegeDowngradeError Text
    | InvalidCapability Text
    | AuthorizationError Text
    deriving (Show, Eq)

instance Exception PrivilegeError

-- | Basic request message type
data RequestMessage = RequestMessage {
    reqId :: Int,
    reqTag :: RequestTag,
    reqPayload :: Aeson.Value,
    reqCapabilities :: Set DaemonCapability,  -- Required capabilities
    reqPrivilege :: RequestPrivilege,         -- Privilege requirement
    reqAuth :: Maybe AuthToken                -- Authentication token
} deriving (Show, Eq)

-- | Basic response message type
data ResponseMessage = ResponseMessage {
    respId :: Int,
    respTag :: ResponseTag,
    respPayload :: Aeson.Value,
    respRequiresAuth :: Bool
} deriving (Show, Eq)

-- | Authentication request with privilege information
data AuthRequest = AuthRequest {
    authVersion :: ProtocolVersion,
    authUser :: Text,
    authToken :: Text,
    authRequestedTier :: PrivilegeTier  -- Requested privilege tier
} deriving (Show, Eq, Generic)

-- | Authentication result with privilege evidence
data AuthResult
    = AuthAccepted UserId AuthToken (Set DaemonCapability)
    | AuthRejected Text
    deriving (Show, Eq, Generic)

-- | Verify if a request has the necessary capabilities
verifyCapabilities :: SPrivilegeTier t -> Set DaemonCapability -> Either PrivilegeError ()
verifyCapabilities st capabilities =
    case fromSing st of
        -- Daemon context can perform any operation
        'Daemon -> Right ()

        -- Builder context has limited capabilities
        'Builder ->
            if any restrictedCapability (Set.toList capabilities)
                then Left $ InsufficientPrivileges $
                    "Operation requires daemon privileges: " <>
                    T.intercalate ", " (map (T.pack . show) $
                                        filter restrictedCapability $
                                        Set.toList capabilities)
                else Right ()
  where
    restrictedCapability :: DaemonCapability -> Bool
    restrictedCapability StoreAccess = True
    restrictedCapability SandboxCreation = True
    restrictedCapability GarbageCollection = True
    restrictedCapability DerivationRegistration = True
    restrictedCapability _ = False
```

```haskell
-- | Check if a request can be performed with given privilege tier
checkPrivilegeRequirement :: SPrivilegeTier t -> RequestPrivilege -> Either PrivilegeError ()
checkPrivilegeRequirement st reqPriv =
    case (fromSing st, reqPriv) of
        ('Daemon', _) ->
            -- Daemon can perform any operation
            Right ()
        ('Builder', PrivilegedRequest) ->
            -- Builder can't perform privileged operations
            Left $ InsufficientPrivileges
                "This operation requires daemon privileges"
        ('Builder', UnprivilegedRequest) ->
            -- Builder can perform unprivileged operations
            Right ()

-- | Send a request with privilege checking
sendRequest :: forall t. SingI t =>
              SPrivilegeTier t -> ProtocolHandle -> DaemonRequest -> AuthToken ->
              IO (Either PrivilegeError Int)
sendRequest st handle req authToken = do
    -- Get request capabilities and privilege requirement
    let capabilities = requestCapabilities req
        privReq = requestPrivilegeRequirement req

    -- Verify capabilities
    case verifyCapabilities st capabilities of
        Left err -> return $ Left err
        Right () ->
            -- Check privilege requirement
            case checkPrivilegeRequirement st privReq of
                Left err -> return $ Left err
                Right () -> do
                    -- Convert to protocol message
                    let reqMsg = RequestMessage {
                            reqId = 0,  -- Will be assigned by sendMessage
                            reqTag = requestTypeToTag req,
                            reqPayload = Aeson.toJSON req,
                            reqCapabilities = capabilities,
                            reqPrivilege = privReq,
                            reqAuth = Just authToken
                        }

                    -- Create appropriate message based on privilege context
                    case fromSing st of
                        'Daemon -> do
                            -- Privileged message
                            let msg = PrivilegedMsg reqMsg
                            -- Send and return ID
                            Right <$> sendMessage handle msg

                        'Builder -> do
                            -- Unprivileged message
                            let msg = UnprivilegedMsg reqMsg
                            -- Send and return ID
                            Right <$> sendMessage handle msg

-- | Determine capabilities required for a request
requestCapabilities :: DaemonRequest -> Set DaemonCapability
requestCapabilities req = case req of
    -- Store operations
    StoreAddRequest{} -> Set.singleton StoreAccess
    StoreVerifyRequest{} -> Set.singleton StoreQuery
    StorePathRequest{} -> Set.singleton StoreQuery
    StoreListRequest -> Set.singleton StoreQuery

    -- Build operations
```

```haskell
    BuildRequest{} -> Set.singleton DerivationBuild
    EvalRequest{} -> Set.singleton DerivationBuild
    BuildDerivationRequest{} -> Set.singleton DerivationBuild
    BuildStatusRequest{} -> Set.singleton BuildQuery
    CancelBuildRequest{} -> Set.singleton BuildQuery

    -- GC operations
    GCRequest{} -> Set.singleton GarbageCollection

    -- Derivation operations
    DerivationStoreRequest{} -> Set.fromList [DerivationRegistration, StoreAccess]
    DerivationQueryRequest{} -> Set.singleton StoreQuery

    -- Administrative operations
    StatusRequest -> Set.singleton BuildQuery
    ConfigRequest -> Set.singleton BuildQuery
    ShutdownRequest -> Set.singleton GarbageCollection

    -- Default
    _ -> Set.singleton BuildQuery

-- | Determine privilege requirement for a request
requestPrivilegeRequirement :: DaemonRequest -> RequestPrivilege
requestPrivilegeRequirement req = case req of
    -- Operations requiring daemon privileges
    StoreAddRequest{} -> PrivilegedRequest
    GCRequest{} -> PrivilegedRequest
    DerivationStoreRequest{} -> PrivilegedRequest
    ShutdownRequest -> PrivilegedRequest

    -- Operations that can be done from either context
    BuildRequest{} -> UnprivilegedRequest
    EvalRequest{} -> UnprivilegedRequest
    BuildDerivationRequest{} -> UnprivilegedRequest
    BuildStatusRequest{} -> UnprivilegedRequest
    StoreVerifyRequest{} -> UnprivilegedRequest
    StoreListRequest -> UnprivilegedRequest
    StatusRequest -> UnprivilegedRequest

    -- Default to privileged for safety
    _ -> PrivilegedRequest

-- | Protocol handle type for managing connections
data ProtocolHandle = ProtocolHandle {
    protocolSocket :: Socket,
    protocolLock :: MVar (),  -- For thread safety
    protocolPrivilege :: PrivilegeTier -- Track privilege level of connection
}

-- | Create a protocol handle with privilege information
createHandle :: Socket -> PrivilegeTier -> IO ProtocolHandle
createHandle sock tier = do
    lock <- newMVar ()
    return $ ProtocolHandle sock lock tier

-- | Send a message through the protocol handle
sendMessage :: ProtocolHandle -> Message p t -> IO Int
sendMessage handle msg = undefined -- Implement actual sending logic

-- Remaining implementations would follow...
```

2.11 src/Ten/Daemon/Client.hs

Ten/Daemon/Client.hs Changes

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE FlexibleContexts #-}

module Ten.Daemon.Client (
    -- Socket management with privilege awareness
    connectToDaemon,
    disconnectFromDaemon,
    getDefaultSocketPath,
    withDaemonConnection,

    -- Privilege-aware client communication
    sendRequest,
    receiveResponse,
    sendRequestSync,

    -- Status checking
    isDaemonRunning,
    getDaemonStatus,

    -- Context-aware build operations
    buildFile,
    evalFile,
    buildDerivation,
    cancelBuild,
    getBuildStatus,
    getBuildOutput,
    listBuilds,

    -- Privilege-aware store operations
    addFileToStore,
    verifyStorePath,
    getStorePathForFile,
    listStore,

    -- Derivation operations with proper context
    storeDerivation,
    retrieveDerivation,
    queryDerivationForOutput,
    queryOutputsForDerivation,
    listDerivations,
    getDerivationInfo,

    -- Privilege-checked GC operations
    collectGarbage,
    getGCStatus,
    addGCRoot,
    removeGCRoot,
    listGCRoots,

    -- Daemon management
    startDaemonIfNeeded,
    shutdownDaemon,
    getDaemonConfig,

    -- Authentication types re-exports
    UserCredentials(..),

    -- Internal utilities exposed for testing
```

```
    createSocketAndConnect,
    readResponseWithTimeout,
    encodeRequest,
    decodeResponse
) where

import Control.Concurrent (forkIO, ThreadId, threadDelay, myThreadId, killThread)
import Control.Concurrent.MVar (MVar, newEmptyMVar, putMVar, takeMVar, newMVar, readMVar)
import Control.Exception (catch, finally, bracketOnError, bracket, throwIO, SomeException, try, IOException)
import Control.Monad (void, when, forever, unless)
import qualified Data.Aeson as Aeson
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy as LBS
import Data.ByteString.Builder (toLazyByteString, word32BE, byteString)
import qualified Data.ByteString.Char8 as BC
import Data.IORef (IORef, newIORef, atomicModifyIORef', readIORef, writeIORef)
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import Data.Time.Clock (UTCTime, getCurrentTime, addUTCTime, diffUTCTime)
import Data.Word (Word32)
import Network.Socket (Socket, Family(..), SocketType(..), SockAddr(..), socket, connect, close, socketToFd,
↪  socketToHandle)
import Network.Socket.ByteString (sendAll, recv)
import System.Directory (doesFileExist, createDirectoryIfMissing, getHomeDirectory, getXdgDirectory,
↪  XdgDirectory(..), findExecutable)
import System.Environment (lookupEnv)
import System.Exit (ExitCode(..))
import System.FilePath ((</>), takeDirectory)
import System.IO (Handle, IOMode(..), hClose, hFlush, hPutStrLn, stderr, hPutStr, BufferMode(..), hSetBuffering)
import System.Process (createProcess, proc, waitForProcess, StdStream(..), CreateProcess(..))
import System.Timeout (timeout)

import Ten.Core
import Ten.Daemon.Protocol
import Ten.Daemon.Auth (UserCredentials(..))
import Ten.Daemon.Config (getDefaultSocketPath)
import Ten.Derivation (Derivation, serializeDerivation, deserializeDerivation)

-- | Connection state for daemon communication
data ConnectionState = ConnectionState {
    csSocket :: Socket,                     -- ^ Socket connected to daemon
    csHandle :: Handle,                     -- ^ Handle for socket I/O
    csUserId :: UserId,                     -- ^ Authenticated user ID
    csToken :: AuthToken,                   -- ^ Authentication token
    csRequestMap :: TVar (Map Int (MVar Response)), -- ^ Map of pending requests
    csNextReqId :: TVar Int,                -- ^ Next request ID
    csReaderThread :: ThreadId,             -- ^ Thread ID of the response reader
    csShutdown :: TVar Bool,                -- ^ Flag to indicate connection shutdown
    csLastError :: TVar (Maybe BuildError), -- ^ Last error encountered
    csCapabilities :: Set DaemonCapability, -- ^ Granted capabilities from auth
    csPrivilegeTier :: PrivilegeTier        -- ^ Always 'Builder for client
}

-- | Daemon connection type with privilege context
data DaemonConnection = DaemonConnection {
    connSocket :: Socket,
    connUserId :: UserId,
    connAuthToken :: AuthToken,
    connState :: ConnectionState,
    -- Builder privilege singleton for runtime evidence
    connPrivEvidence :: SPrivilegeTier 'Builder
}
```

```haskell
-- | Connect to the Ten daemon - always in Builder context
connectToDaemon :: FilePath -> UserCredentials -> IO (Either BuildError DaemonConnection)
connectToDaemon socketPath credentials = try $ do
    -- Check if daemon is running
    running <- isDaemonRunning socketPath

    -- If not running, try to start it if autostart is enabled
    unless running $ do
        startResult <- startDaemonIfNeeded socketPath
        case startResult of
            Left err -> throwIO err
            Right _ ->
                -- Brief pause to allow daemon to initialize
                threadDelay 500000 -- 0.5 seconds

    -- Create socket and connect
    (sock, handle) <- createSocketAndConnect socketPath

    -- Initialize request tracking
    requestMap <- newTVarIO Map.empty
    nextReqId <- newTVarIO 1
    shutdownFlag <- newTVarIO False
    lastError <- newTVarIO Nothing

    -- Authenticate with the daemon
    let authReq = AuthRequest {
            authVersion = currentProtocolVersion,
            authUser = username credentials,
            authToken = token credentials,
            authRequestedTier = 'Builder -- Always request Builder tier
        }

    -- Encode auth request
    let reqBS = serializeMessage (AuthRequestMsg authReq)

    -- Send auth request
    BS.hPut handle reqBS
    hFlush handle

    -- Read auth response
    respBS <- readMessageWithTimeout handle 5000000 -- 5 seconds timeout

    -- Parse and handle the response
    case deserializeMessage respBS of
        Left err ->
            throwIO $ AuthError $ "Authentication failed: " <> err

        Right (AuthResponseMsg (AuthAccepted userId authToken capabilities)) -> do
            -- Set up proper handle buffering
            hSetBuffering handle (BlockBuffering Nothing)

            -- Start background thread to read responses
            readerThread <- forkIO $ responseReaderThread handle requestMap shutdownFlag lastError

            -- Create connection state
            let connState = ConnectionState {
                    csSocket = sock,
                    csHandle = handle,
                    csUserId = userId,
                    csToken = authToken,
                    csRequestMap = requestMap,
                    csNextReqId = nextReqId,
                    csReaderThread = readerThread,
                    csShutdown = shutdownFlag,
                    csLastError = lastError,
                    csCapabilities = capabilities,
```

```
                    csPrivilegeTier = 'Builder -- Always 'Builder for client
                }

            -- Get singleton evidence for Builder context
            let privilegeEvidence = sing @'Builder

            -- Return connection object with privilege evidence
            return $ DaemonConnection sock userId authToken connState privilegeEvidence

        Right (AuthResponseMsg (AuthRejected reason)) ->
            throwIO $ AuthError $ "Authentication rejected: " <> reason

-- | Build a file using the daemon
buildFile :: DaemonConnection -> FilePath -> IO (Either BuildError BuildResult)
buildFile conn filePath = do
    -- Check file existence
    fileExists <- doesFileExist filePath
    unless fileExists $
        return $ Left $ InputNotFound filePath

    -- Read file content
    content <- BS.readFile filePath

    -- Create build request - works in Builder context
    let request = BuildRequest
            { buildFilePath = T.pack filePath
            , buildFileContent = Just content
            , buildOptions = defaultBuildRequestInfo
            }

    -- Send request with privilege evidence and wait for response
    respResult <- sendRequestSync (connPrivEvidence conn) conn request (120 * 1000000) -- 120 seconds timeout

    -- Process response
    case respResult of
        Left err ->
            return $ Left err

        Right (BuildResponse result) ->
            return $ Right result

        Right resp ->
            return $ Left $ DaemonError $
                "Invalid response type for build request: " <> T.pack (show resp)

-- | Execute a GC operation (requires Daemon privileges)
collectGarbage :: DaemonConnection -> Bool -> IO (Either BuildError GCStats)
collectGarbage conn force = do
    -- Create GC request
    let request = GCRequest
            { gcForce = force
            }

    -- Send request with privilege evidence
    -- Note: This will fail because GC requires Daemon privileges
    -- and client only has Builder privileges
    respResult <- sendRequestSync (connPrivEvidence conn) conn request (300 * 1000000) -- 5 minutes timeout

    -- Process response
    case respResult of
        Left err ->
            return $ Left err

        Right (GCResponse stats) ->
            return $ Right stats

        Right resp ->
```

```
            return $ Left $ DaemonError $
                "Invalid response type for GC request: " <> T.pack (show resp)

-- | Send a request with privilege checking and wait for response
sendRequestSync :: SPrivilegeTier t -> DaemonConnection -> DaemonRequest -> Int ->
                   IO (Either BuildError DaemonResponse)
sendRequestSync st conn request timeoutMicros = do
    -- Get request capabilities
    let capabilities = requestCapabilities request

    -- Verify capabilities against privilege tier
    case verifyCapabilities st capabilities of
        Left err ->
            -- Return privilege error
            return $ Left $ PrivilegeError $ T.pack $ show err

        Right () -> do
            -- Send request and get ID
            reqIdResult <- sendRequest st conn request (connAuthToken conn)

            case reqIdResult of
                Left err ->
                    -- Return privilege error
                    return $ Left $ PrivilegeError $ T.pack $ show err

                Right reqId -> do
                    -- Wait for response
                    respResult <- receiveResponse conn reqId timeoutMicros

                    -- Process the response
                    case respResult of
                        Left err ->
                            return $ Left err

                        Right resp ->
                            -- Convert to Response type
                            case responseToResponseData resp of
                                Left err ->
                                    return $ Left $ DaemonError $ "Failed to decode response: " <> err
                                Right respData ->
                                    return $ Right respData

-- | Store a derivation - requires Daemon privileges for direct store access
storeDerivation :: DaemonConnection -> Derivation -> IO (Either BuildError StorePath)
storeDerivation conn derivation = do
    -- Serialize the derivation
    let serialized = serializeDerivation derivation

    -- Create store derivation request
    let request = StoreDerivationRequest
            { derivationContent = serialized
            }

    -- Send request with privilege evidence
    respResult <- sendRequestSync (connPrivEvidence conn) conn request (30 * 1000000) -- 30 seconds timeout

    -- Process response
    case respResult of
        Left err ->
            return $ Left err

        Right (DerivationStoredResponse path) ->
            return $ Right path

        Right resp ->
            return $ Left $ DaemonError $
                "Invalid response type for store derivation request: " <> T.pack (show resp)
```

```haskell
-- | Add a file to the store - requires Daemon privileges
addFileToStore :: DaemonConnection -> FilePath -> IO (Either BuildError StorePath)
addFileToStore conn filePath = do
    -- Check file existence
    fileExists <- doesFileExist filePath
    unless fileExists $
        return $ Left $ InputNotFound filePath

    -- Read file content
    content <- BS.readFile filePath

    -- Create store add request
    let request = StoreAddRequest
            { storeAddPath = T.pack filePath
            , storeAddContent = content
            }

    -- Send request with privilege evidence
    respResult <- sendRequestSync (connPrivEvidence conn) conn request (60 * 1000000) -- 60 seconds timeout

    -- Process response
    case respResult of
        Left err ->
            return $ Left err

        Right (StoreAddResponse path) ->
            return $ Right path

        Right resp ->
            return $ Left $ DaemonError $
                "Invalid response type for store add request: " <> T.pack (show resp)

-- | Verify a store path - available in Builder context
verifyStorePath :: DaemonConnection -> StorePath -> IO (Either BuildError Bool)
verifyStorePath conn path = do
    -- Create verify request
    let request = StoreVerifyRequest
            { storeVerifyPath = storePathToText path
            }

    -- Send request with privilege evidence
    respResult <- sendRequestSync (connPrivEvidence conn) conn request (30 * 1000000) -- 30 seconds timeout

    -- Process response
    case respResult of
        Left err ->
            return $ Left err

        Right (StoreVerifyResponse valid) ->
            return $ Right valid

        Right resp ->
            return $ Left $ DaemonError $
                "Invalid response type for store verify request: " <> T.pack (show resp)

-- | Background thread to read and dispatch responses
responseReaderThread :: Handle -> TVar (Map Int (MVar Response)) -> TVar Bool ->
                        TVar (Maybe BuildError) -> IO ()
responseReaderThread handle requestMap shutdownFlag lastError = do
    let loop = do
            -- Check if we should shut down
            shutdown <- readTVar shutdownFlag
            unless shutdown $ do
                -- Try to read a message with error handling
                result <- try $ readMessage handle
                case result of
```

```
                    Left (e :: SomeException) -> do
                        -- Socket error, write to lastError and exit thread
                        writeTVar lastError $ Just $ DaemonError $
                            "Connection error: " <> T.pack (show e)
                        return ()

                    Right msgBS -> do
                        -- Process message if valid
                        case deserializeMessage msgBS of
                            Left err -> do
                                -- Parsing error, write to lastError and continue
                                writeTVar lastError $ Just $ DaemonError $
                                    "Protocol error: " <> err
                                loop

                            Right (ResponseMsgWrapper (ResponseMsg reqId resp)) -> do
                                -- Look up request
                                reqMap <- readTVar requestMap
                                case Map.lookup reqId reqMap of
                                    Nothing ->
                                        -- Unknown request ID, ignore and continue
                                        loop

                                    Just respVar -> do
                                        -- Check privilege requirements for response
                                        let privRequirement = responsePrivilegeRequirement resp

                                        -- Builder context can only receive unprivileged responses
                                        case privRequirement of
                                            PrivilegedResponse -> do
                                                -- Cannot receive privileged response in builder context
                                                writeTVar lastError $ Just $ PrivilegeError
                                                    "Received privileged response in builder context"
                                                -- Still deliver to unblock waiting thread
                                                putMVar respVar resp
                                                loop

                                            UnprivilegedResponse -> do
                                                -- Deliver response
                                                putMVar respVar resp
                                                loop

                            Right _ ->
                                -- Other message type, ignore and continue
                                loop

    -- Start the loop and handle exceptions
    atomically loop `catch` (\(e :: SomeException) -> do
        -- Store the error
        atomically $ writeTVar lastError $ Just $ DaemonError $
            "Connection error: " <> T.pack (show e)
        -- Continue loop if the socket is still open
        shouldContinue <- atomically $ not <$> readTVar shutdownFlag
        when shouldContinue loop)

-- | Add helper to determine privilege requirement for response types
responsePrivilegeRequirement :: Response -> ResponsePrivilege
responsePrivilegeRequirement resp = case resp of
    -- Responses containing privileged information
    StoreAddResponse{} -> PrivilegedResponse
    GCResponse{} -> PrivilegedResponse

    -- Responses that can be received in any context
    BuildResponse{} -> UnprivilegedResponse
    BuildStatusResponse{} -> UnprivilegedResponse
    StoreVerifyResponse{} -> UnprivilegedResponse
    StoreListResponse{} -> UnprivilegedResponse
```

```haskell
    -- By default, treat as unprivileged
    _ -> UnprivilegedResponse

-- | Tag for response privilege requirements
data ResponsePrivilege =
    PrivilegedResponse   -- Can only be received in daemon context
  | UnprivilegedResponse -- Can be received in any context
  deriving (Show, Eq)

-- Rest of implementation would follow...
```

2.12 src/Ten/Daemon/Server.hs

Ten/Daemon/Server.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE LambdaCase #-}

module Ten.Daemon.Server where

import Control.Concurrent (forkIO, ThreadId)
import Control.Concurrent.STM
import Control.Exception (bracket, finally, try, SomeException)
import Control.Monad (unless, when, void)
import Control.Monad.Reader
import Control.Monad.Except
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T
import Data.Singletons
import Network.Socket

import Ten.Core
import Ten.Daemon.Protocol
import Ten.Daemon.State
import Ten.Daemon.Auth
import Ten.Daemon.Config

-- | Server control information with proper privilege tier tagging
data ServerControl (t :: PrivilegeTier) = ServerControl {
    scThread :: ThreadId,              -- ^ Main server thread
    scClients :: TVar (Map ClientId ClientInfo),  -- ^ Connected clients
    scShutdown :: TVar Bool,           -- ^ Shutdown flag
    scState :: DaemonState t,          -- ^ Daemon state with privilege evidence
    scConfig :: DaemonConfig           -- ^ Daemon configuration
}

-- | Client information with privilege tier information
data ClientInfo = ClientInfo {
    ciSocket :: Socket,                -- ^ Client socket
    ciUserId :: UserId,                -- ^ Authenticated user ID
    ciPrivilegeTier :: PrivilegeTier,  -- ^ Privilege level granted
    ciAuthToken :: AuthToken,          -- ^ Authentication token
    ciPermissions :: Set Permission    -- ^ Granted permissions
}

-- | Start the daemon server with proper privilege evidence
```

```haskell
startServer ::
    SPrivilegeTier 'Daemon ->      -- ^ Singleton evidence for daemon privilege
    Socket ->                       -- ^ Listening socket
    DaemonState 'Daemon ->         -- ^ Daemon state (privileged)
    DaemonConfig ->                -- ^ Configuration
    TenM 'Build 'Daemon (ServerControl 'Daemon)
startServer st sock state config = TenM $ \sp _ -> do
    -- Initialize client tracking with privilege checks
    clients <- newTVarIO Map.empty
    shutdownFlag <- newTVarIO False

    -- Start listener thread in privileged context
    serverThread <- forkIO $ do
        let runServer = acceptClients sock clients shutdownFlag state config

        -- Run with daemon privileges
        result <- runTen runServer
                  (initDaemonEnv (daemonTmpDir config) (daemonStorePath config) (Just "daemon"))
                  (initBuildState Build (BuildIdFromInt 0) 'Daemon)

        case result of
            Left err -> putStrLn $ "Server error: " ++ show err
            Right _ -> return ()

    -- Return the server control with privilege evidence
    return $ ServerControl {
        scThread = serverThread,
        scClients = clients,
        scShutdown = shutdownFlag,
        scState = state,
        scConfig = config
    }

-- | Stop the server with proper privilege validation
stopServer ::
    SPrivilegeTier 'Daemon ->        -- ^ Privilege evidence
    ServerControl 'Daemon ->         -- ^ Server control
    TenM 'Build 'Daemon ()
stopServer st control = TenM $ \sp _ -> do
    -- Set shutdown flag
    atomically $ writeTVar (scShutdown control) True

    -- Get client info for cleanup
    clientMap <- atomically $ readTVar (scClients control)

    -- Close all client connections
    forM_ (Map.elems clientMap) $ \ci -> do
        -- Send shutdown notification with privilege check
        sendShutdownNotice st (ciSocket ci)

        -- Close the socket
        close (ciSocket ci)

    -- Log shutdown
    logMessage st "Daemon server shutting down"

-- | Accept client connections with proper privilege validation
acceptClients ::
    SPrivilegeTier 'Daemon ->      -- ^ Privilege evidence
    Socket ->                       -- ^ Listening socket
    TVar (Map ClientId ClientInfo) -> -- ^ Client tracking
    TVar Bool ->                    -- ^ Shutdown flag
    DaemonState 'Daemon ->         -- ^ Daemon state
    DaemonConfig ->                -- ^ Configuration
    TenM 'Build 'Daemon ()
acceptClients st sock clients shutdownFlag state config = TenM $ \sp _ -> do
    let acceptLoop = do
```

```
            -- Check shutdown flag
            shouldShutdown <- readTVarIO shutdownFlag
            unless shouldShutdown $ do
                -- Accept client connection (with privilege check)
                (clientSock, clientAddr) <- accept sock

                -- Set socket options with privilege
                setSocketOption clientSock ReuseAddr 1

                -- Validate connection rate limits
                allowed <- checkConnectionRateLimit clientAddr

                if not allowed
                    then do
                        -- Rate limit exceeded, close connection
                        close clientSock
                        acceptLoop
                    else do
                        -- Fork client handler thread with proper privilege
                        clientThread <- forkIO $ do
                            handleClient st clientSock clientAddr clients state config

                        -- Continue accepting
                        acceptLoop

    -- Start accept loop
    acceptLoop

-- | Handle a client connection with proper privilege validation
handleClient ::
    SPrivilegeTier 'Daemon ->      -- ^ Privilege evidence
    Socket ->                       -- ^ Client socket
    SockAddr ->                     -- ^ Client address
    TVar (Map ClientId ClientInfo) -> -- ^ Client tracking
    DaemonState 'Daemon ->         -- ^ Daemon state
    DaemonConfig ->                -- ^ Configuration
    TenM 'Build 'Daemon ()
handleClient st clientSock clientAddr clients state config = TenM $ \sp _ -> do
    -- Perform authentication protocol (with privilege boundary check)
    authResult <- authenticateClient st clientSock config

    case authResult of
        Left err -> do
            -- Authentication failed, send error and close
            sendAuthFailure st clientSock err
            close clientSock

        Right (userId, authToken, clientPerm, privTier) -> do
            -- Successfully authenticated

            -- Create client info record
            let clientInfo = ClientInfo {
                    ciSocket = clientSock,
                    ciUserId = userId,
                    ciPrivilegeTier = privTier,
                    ciAuthToken = authToken,
                    ciPermissions = clientPerm
                }

            -- Generate client ID
            clientId <- randomClientId

            -- Register client
            atomically $ modifyTVar' clients $ Map.insert clientId clientInfo

            -- Process client requests based on privilege level
            case privTier of
```

```
                'Daemon ->
                    -- Handle daemon-privileged requests
                    processClientRequests st clientSock userId clientPerm state config

                'Builder ->
                    -- Handle builder-privileged requests
                    processClientRequestsBuilder st clientSock userId clientPerm state config

            -- Remove client when done
            atomically $ modifyTVar' clients $ Map.delete clientId

-- | Process requests from a daemon-privileged client
processClientRequests ::
    SPrivilegeTier 'Daemon ->        -- ^ Privilege evidence
    Socket ->                         -- ^ Client socket
    UserId ->                         -- ^ User ID
    Set Permission ->                 -- ^ Permissions
    DaemonState 'Daemon ->           -- ^ Daemon state
    DaemonConfig ->                   -- ^ Configuration
    TenM 'Build 'Daemon ()
processClientRequests st clientSock userId permissions state config = TenM $ \sp _ -> do
    let processLoop = do
            -- Read request with timeout
            mRequest <- readRequestWithTimeout clientSock 30000000

            case mRequest of
                Nothing ->
                    return () -- Timeout or error, exit loop

                Just (request, reqId) -> do
                    -- Validate request against permissions with privilege evidence
                    allowed <- withDaemonPermission st request permissions

                    if not allowed
                        then do
                            -- Permission denied
                            sendResponse clientSock reqId $
                                ErrorResponse "Permission denied"
                            processLoop
                        else do
                            -- Process the request in daemon context
                            response <- runDaemonRequest st request state

                            -- Send the response
                            sendResponse clientSock reqId response

                            -- Continue processing
                            processLoop

    -- Run the request processing loop
    processLoop `catch` \(e :: SomeException) ->
        -- Log error and terminate gracefully
        logError st $ "Error handling client: " <> T.pack (show e)

-- | Process requests from a builder-privileged client
-- Similar to processClientRequests but with Builder privilege context
processClientRequestsBuilder ::
    SPrivilegeTier 'Daemon ->        -- ^ Daemon privilege evidence
    Socket ->                         -- ^ Client socket
    UserId ->                         -- ^ User ID
    Set Permission ->                 -- ^ Permissions
    DaemonState 'Daemon ->           -- ^ Daemon state
    DaemonConfig ->                   -- ^ Configuration
    TenM 'Build 'Daemon ()
processClientRequestsBuilder st clientSock userId permissions state config = TenM $ \sp _ -> do
    -- Create builder privilege singleton
    let builderSt = SBuilder -- This is where we drop privileges for the client
```

```haskell
    -- Similar to processClientRequests but using builder privilege context
    let processLoop = do
            -- Read request
            mRequest <- readRequestWithTimeout clientSock 30000000

            case mRequest of
                Nothing ->
                    return () -- Timeout or error, exit loop

                Just (request, reqId) -> do
                    -- Validate request against permissions with builder evidence
                    allowed <- withBuilderPermission builderSt request permissions

                    if not allowed
                        then do
                            -- Permission denied
                            sendResponse clientSock reqId $
                                ErrorResponse "Permission denied"
                            processLoop
                        else do
                            -- Process the request in builder context
                            -- Note how we drop privilege here by using transitionPrivilege
                            let builderAction = runBuilderRequest builderSt request state
                            let daemonAction = transitionPrivilege builderAction

                            response <- daemonAction

                            -- Send the response
                            sendResponse clientSock reqId response

                            -- Continue processing
                            processLoop

    -- Run the builder request processing loop
    processLoop `catch` \(e :: SomeException) ->
        logError st $ "Error handling builder client: " <> T.pack (show e)

-- | Validate if a request can be processed with daemon privileges
withDaemonPermission ::
    SPrivilegeTier 'Daemon ->           -- ^ Privilege evidence
    Request ->                           -- ^ Request to validate
    Set Permission ->                    -- ^ Granted permissions
    TenM 'Build 'Daemon Bool
withDaemonPermission st request permissions = TenM $ \sp _ -> do
    -- Determine the required permission and privilege level for this request
    let requiredPermission = getRequiredPermission request
    let requiredPrivilege = getRequiredPrivilege request

    -- Check if the client has the required permission
    let hasPermission = requiredPermission `Set.member` permissions

    -- Check if the privilege level is sufficient (already 'Daemon here)
    let hasPrivilege = requiredPrivilege == Daemon

    -- All checks must pass
    return $ hasPermission && hasPrivilege

-- | Validate if a request can be processed with builder privileges
withBuilderPermission ::
    SPrivilegeTier 'Builder ->           -- ^ Privilege evidence
    Request ->                           -- ^ Request to validate
    Set Permission ->                    -- ^ Granted permissions
    TenM 'Build 'Builder Bool
withBuilderPermission st request permissions = TenM $ \sp _ -> do
    -- Determine the required permission and privilege level for this request
    let requiredPermission = getRequiredPermission request
```

```
    let requiredPrivilege = getRequiredPrivilege request

    -- Check if the client has the required permission
    let hasPermission = requiredPermission `Set.member` permissions

    -- Check if the permission can be used in Builder context
    -- Some permissions require Daemon privileges
    let validForBuilder = not (requiresDaemonPrivilege requiredPermission)

    -- For Builder tier, also verify if the requested operation is allowed in Builder context
    let sufficientPrivilege = case requiredPrivilege of
            Builder -> True      -- Builder privilege is sufficient
            Daemon -> False      -- Daemon privilege required, can't proceed

    -- All checks must pass
    return $ hasPermission && validForBuilder && sufficientPrivilege

-- | Check if a permission requires Daemon privileges
requiresDaemonPrivilege :: Permission -> Bool
requiresDaemonPrivilege = \case
    PermModifyStore -> True        -- Store modifications require Daemon
    PermRunGC -> True              -- GC operations require Daemon
    PermShutdown -> True           -- Shutdown requires Daemon
    PermManageUsers -> True        -- User management requires Daemon
    _ -> False                     -- Other permissions can work with Builder

-- | Process a request in daemon privilege context
runDaemonRequest ::
    SPrivilegeTier 'Daemon ->             -- ^ Privilege evidence
    Request ->                            -- ^ Request to process
    DaemonState 'Daemon ->                -- ^ Daemon state
    TenM 'Build 'Daemon Response
runDaemonRequest st request state = TenM $ \sp _ -> do
    -- Dispatch to appropriate handler based on request type with privilege evidence
    case request of
        BuildRequest path content options ->
            handleBuildRequest st path content options state

        QueryStoreRequest path ->
            handleStoreQueryRequest st path state

        ModifyStoreRequest path content ->
            handleStoreModifyRequest st path content state

        GCRequest force ->
            handleGCRequest st force state

        -- Other request handlers with proper privilege handling
        _ ->
            -- Default case - unknown request
            return $ ErrorResponse "Unknown request type"

-- | Process a request in builder privilege context
runBuilderRequest ::
    SPrivilegeTier 'Builder ->             -- ^ Privilege evidence
    Request ->                             -- ^ Request to process
    DaemonState 'Builder ->                -- ^ Daemon state
    TenM 'Build 'Builder Response
runBuilderRequest st request state = TenM $ \sp _ -> do
    -- Dispatch to appropriate handler based on request type
    -- Note that these are builder-safe operations only
    case request of
        BuildRequest path content options ->
            handleBuilderBuildRequest st path content options state

        QueryStoreRequest path ->
            handleBuilderStoreQueryRequest st path state
```

```haskell
        -- Explicitly reject operations requiring Daemon privilege
        ModifyStoreRequest {} ->
            return $ ErrorResponse "Store modification requires daemon privileges"

        GCRequest {} ->
            return $ ErrorResponse "GC operations require daemon privileges"

        -- Other request handlers
        _ ->
            return $ ErrorResponse "Unknown or prohibited request type"

-- | Handle a build request with daemon privileges
handleBuildRequest ::
    SPrivilegeTier 'Daemon ->             -- ^ Privilege evidence
    Text ->                               -- ^ Path to build
    Maybe BuildContent ->                 -- ^ Optional build content
    BuildOptions ->                       -- ^ Build options
    DaemonState 'Daemon ->                -- ^ Daemon state
    TenM 'Build 'Daemon Response
handleBuildRequest st path content options state = TenM $ \sp _ -> do
    -- Process the build with full daemon privileges
    -- Validate input
    validPath <- validatePath st path
    unless validPath $
        return $ ErrorResponse "Invalid build path"

    -- Get content
    buildContent <- case content of
        Just c -> return c
        Nothing -> readBuildFile st path

    -- Parse build file
    derivation <- parseBuildFile st buildContent

    -- Register the build
    buildId <- registerBuild st state derivation options

    -- If async, start build in background
    if asyncBuild options
        then do
            -- Fork with proper privilege tracking
            void $ forkIO $ do
                processBuild st state buildId derivation

            -- Return immediate response
            return $ BuildStartedResponse buildId
        else do
            -- Process build synchronously
            result <- processBuild st state buildId derivation

            -- Return complete result
            return $ BuildResultResponse result

-- | Handle a store modification request (requires daemon privileges)
handleStoreModifyRequest ::
    SPrivilegeTier 'Daemon ->             -- ^ Privilege evidence
    Text ->                               -- ^ Path
    BuildContent ->                       -- ^ Content
    DaemonState 'Daemon ->                -- ^ Daemon state
    TenM 'Build 'Daemon Response
handleStoreModifyRequest st path content state = TenM $ \sp _ -> do
    -- This is a privileged operation requiring Daemon context
    -- Add content to store
    storePath <- withStore st $ \st' ->
        addToStore st' (T.unpack path) content
```

```haskell
    -- Register in reachable paths
    markPathAsReachable st state storePath

    -- Return the store path
    return $ StorePathResponse storePath

-- | Handle a GC request (requires daemon privileges)
handleGCRequest ::
    SPrivilegeTier 'Daemon ->        -- ^ Privilege evidence
    Bool ->                          -- ^ Force flag
    DaemonState 'Daemon ->           -- ^ Daemon state
    TenM 'Build 'Daemon Response
handleGCRequest st force state = TenM $ \sp _ -> do
    -- Acquire GC lock
    acquired <- tryAcquireGCLock st state

    if not acquired && not force
        then return $ ErrorResponse "GC already in progress"
        else do
            -- Run GC with proper privilege
            gcStats <- withGCLock st state $ \st' ->
                collectGarbage st'

            -- Return results
            return $ GCResultResponse gcStats

-- | Handle a build request in builder context
handleBuilderBuildRequest ::
    SPrivilegeTier 'Builder ->       -- ^ Privilege evidence
    Text ->                          -- ^ Path to build
    Maybe BuildContent ->            -- ^ Optional build content
    BuildOptions ->                  -- ^ Build options
    DaemonState 'Builder ->          -- ^ Daemon state
    TenM 'Build 'Builder Response
handleBuilderBuildRequest st path content options state = TenM $ \sp _ -> do
    -- Process the build with limited builder privileges
    -- Note: will use daemon service for privileged operations

    -- Validate input
    validPath <- validatePath st path
    unless validPath $
        return $ ErrorResponse "Invalid build path"

    -- Get content
    buildContent <- case content of
        Just c -> return c
        Nothing -> readBuildFile st path

    -- Parse build file (this is allowed in Builder context)
    derivation <- parseBuildFile st buildContent

    -- Register the build via daemon protocol (RPC)
    buildId <- registerBuildViaProtocol st state derivation options

    -- Return immediate response (async only in builder context)
    return $ BuildStartedResponse buildId

-- | Authenticate client with proper privilege accounting
authenticateClient ::
    SPrivilegeTier 'Daemon ->      -- ^ Privilege evidence
    Socket ->                      -- ^ Client socket
    DaemonConfig ->                -- ^ Configuration
    TenM 'Build 'Daemon (Either Text (UserId, AuthToken, Set Permission, PrivilegeTier))
authenticateClient st sock config = TenM $ \sp _ -> do
    -- Read authentication request
    mAuthReq <- readAuthRequest sock
```

```
    case mAuthReq of
        Nothing ->
            return $ Left "Invalid authentication request"

        Just (username, password, requestedTier) -> do
            -- Process authentication with privilege validation
            authResult <- authenticateUser username password

            case authResult of
                Left err ->
                    return $ Left err

                Right (userId, token, permissions) -> do
                    -- Determine privilege tier to grant based on requested tier,
                    -- permissions, and daemon configuration
                    grantedTier <- if requestedTier == Daemon && canGrantDaemonTier permissions config
                                   then return Daemon
                                   else return Builder

                    -- Return auth result with granted privilege tier
                    return $ Right (userId, token, permissions, grantedTier)

-- | Check if a user can be granted Daemon privileges
canGrantDaemonTier :: Set Permission -> DaemonConfig -> Bool
canGrantDaemonTier permissions config =
    -- Daemon privilege can be granted if:
    -- 1. User has admin permission
    -- 2. Daemon allows privilege escalation in config
    PermAdmin `Set.member` permissions && daemonAllowPrivilegeEscalation config

-- | Random ID generation for clients
randomClientId :: IO ClientId
randomClientId = do
    -- Generate a random client ID
    uuid <- genRandomUUID
    return $ ClientId uuid

-- | Utility to send shutdown notice to client
sendShutdownNotice ::
    SPrivilegeTier 'Daemon ->      -- ^ Privilege evidence
    Socket ->                       -- ^ Client socket
    TenM 'Build 'Daemon ()
sendShutdownNotice st sock = TenM $ \sp _ -> do
    -- Create shutdown message
    let msg = createShutdownMessage st

    -- Send to client
    sendAll sock msg
```

2.13 src/Ten/Daemon/Core.hs

Ten/Daemon/Core.hs Changes

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE LambdaCase #-}

module Ten.Daemon.Core where

import Control.Concurrent (forkIO, ThreadId)
import Control.Concurrent.STM
import Control.Exception (bracket, finally, try, SomeException)
```

```haskell
import Control.Monad (unless, when, void)
import Control.Monad.Reader
import Control.Monad.Except
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T
import Data.Singletons
import System.Posix.Types (UserID, GroupID)
import System.Posix.User (getUserEntryForName, getGroupEntryForName, setUserID, setGroupID)
import System.Posix.Process (getProcessID)
import System.Directory (doesFileExist, createDirectoryIfMissing, getPermissions, setPermissions)
import System.FilePath (takeDirectory, (</>))
import System.Process (createProcess, proc, waitForProcess)
import Network.Socket

import Ten.Core
import Ten.Daemon.Protocol
import Ten.Daemon.State
import Ten.Daemon.Config
import Ten.Build (BuildResult(..))
import Ten.Store (initializeStore, createStoreDirectories, verifyStorePath)
import Ten.Sandbox (SandboxConfig(..), defaultSandboxConfig, setupSandbox, teardownSandbox)

-- | Daemon context with privilege tier information
data DaemonContext (t :: PrivilegeTier) = DaemonContext {
    ctxConfig :: DaemonConfig,          -- ^ Daemon configuration
    ctxState :: DaemonState t,          -- ^ Daemon state with privilege tracking
    ctxSocket :: Maybe Socket,          -- ^ Listening socket
    ctxPrivilegeEvidence :: SPrivilegeTier t, -- ^ Runtime privilege evidence
    ctxStartTime :: UTCTime,            -- ^ Daemon start time
    ctxProcessId :: ProcessID,          -- ^ Process ID
    ctxLogHandle :: Handle,             -- ^ Log file handle
    ctxUnprivilegedUserID :: UserID,    -- ^ UID for unprivileged operations
    ctxUnprivilegedGroupID :: GroupID   -- ^ GID for unprivileged operations
}

-- | Start the daemon with proper privilege handling
startDaemon :: DaemonConfig -> IO ()
startDaemon config = do
    -- Check if daemon is already running
    running <- isDaemonRunning (daemonSocketPath config)
    when running $ do
        putStrLn "Daemon is already running"
        exitSuccess

    -- Setup essential directories
    ensureDirectories config

    -- Get unprivileged user/group IDs
    (unprivUid, unprivGid) <- getUnprivilegedIDs config

    -- Decide whether to run in foreground or background
    if daemonForeground config
        then do
            putStrLn "Starting daemon in foreground mode..."
            runDaemon config unprivUid unprivGid
        else do
            putStrLn "Starting daemon in background mode..."
            -- Daemonize
            daemonize $ runDaemon config unprivUid unprivGid

-- | Run the daemon process with privilege management
runDaemon :: DaemonConfig -> UserID -> GroupID -> IO ()
runDaemon config unprivUid unprivGid = do
```

```haskell
    -- Create daemon context with Daemon privilege
    context <- initDaemonContext config unprivUid unprivGid SDaemon

    -- Create listening socket while still privileged
    listenSocket <- createListeningSocket (daemonSocketPath config)

    -- Update context with socket
    let context' = context { ctxSocket = Just listenSocket }

    -- Drop privileges if configured
    when (isJust (daemonUser config)) $ do
        dropPrivilegesRoot context' config

    -- Set up logging
    logHandle <- setupLogging config
    let context'' = context' { ctxLogHandle = logHandle }

    -- Initialize state
    state <- initDaemonState (daemonStateFile config) (daemonMaxJobs config) 100
    let context''' = context'' { ctxState = state }

    -- Verify store integrity with daemon privilege evidence
    verifyStoreIntegrity (ctxPrivilegeEvidence context''') (daemonStorePath config)

    -- Run main daemon loop with privilege boundaries
    finally
        (runDaemonLoop context''')
        (shutdownDaemon context''' logHandle)

-- | Initialize daemon context with proper privilege evidence
initDaemonContext ::
    DaemonConfig ->
    UserID ->
    GroupID ->
    SPrivilegeTier t ->  -- ^ Singleton evidence of privilege
    IO (DaemonContext t)
initDaemonContext config unprivUid unprivGid st = do
    -- Get process ID
    pid <- getProcessID

    -- Record start time
    now <- getCurrentTime

    -- Create context with privilege evidence
    return DaemonContext {
        ctxConfig = config,
        ctxState = error "State not initialized",  -- Will be set later
        ctxSocket = Nothing,                       -- Will be set later
        ctxPrivilegeEvidence = st,                 -- Store privilege evidence
        ctxStartTime = now,
        ctxProcessId = pid,
        ctxLogHandle = error "Log handle not initialized", -- Will be set later
        ctxUnprivilegedUserID = unprivUid,
        ctxUnprivilegedGroupID = unprivGid
    }

-- | Verify store integrity with privilege tracking
verifyStoreIntegrity ::
    SPrivilegeTier 'Daemon ->     -- ^ Privilege evidence
    FilePath ->                   -- ^ Store path
    TenM 'Build 'Daemon ()
verifyStoreIntegrity st storePath = TenM $ \sp _ -> do
    -- Verify store with proper privilege
    result <- withStore st $ \st' ->
        verifyStore st' storePath

    -- Log issues if any
```

```haskell
    case result of
        Left err ->
            logError $ "Store verification issue: " <> T.pack (show err)
        Right _ ->
            logMessage "Store verification successful"

-- | Main daemon loop with privilege boundary checks
runDaemonLoop ::
    DaemonContext 'Daemon ->      -- ^ Daemon context (privileged)
    TenM 'Build 'Daemon ()
runDaemonLoop context = TenM $ \sp st -> do
    -- Create environment with daemon privileges
    let env = ctxConfig context
    let state = ctxState context
    let privilegeEvidence = ctxPrivilegeEvidence context

    -- Handle active builds with proper privilege checks
    builds <- listActiveBuilds privilegeEvidence state

    -- Process each active build
    forM_ builds $ \(buildId, derivation, status) -> do
        -- Check if build is running and needs monitoring
        when (status == BuildRunning) $ do
            -- Monitor build with proper privilege evidence
            monitorBuild privilegeEvidence state buildId

    -- Schedule new builds from queue if slots available
    canSchedule <- hasBuildCapacity privilegeEvidence state
    when canSchedule $ do
        -- Get next build from queue with proper privilege
        nextBuild <- getNextBuildToSchedule privilegeEvidence state

        -- Start build if available
        whenJust nextBuild $ \build -> do
            startBuildProcess privilegeEvidence state (buildId build) (derivation build)

    -- Run periodic maintenance
    runPeriodicMaintenance privilegeEvidence state

    -- Sleep before next iteration
    liftIO $ threadDelay 1000000  -- 1 second

    -- Continue loop
    runDaemonLoop context

-- | Spawn a builder process with proper privilege separation
spawnBuilder ::
    SPrivilegeTier 'Daemon ->       -- ^ Privilege evidence
    Derivation ->                   -- ^ Derivation to build
    BuildId ->                      -- ^ Build ID
    TenM 'Build 'Daemon ProcessHandle
spawnBuilder st derivation buildId = TenM $ \sp _ -> do
    env <- ask

    -- Create sandbox configuration
    let sandboxConfig = defaultSandboxConfig {
            sandboxUser = daemonUser env,
            sandboxGroup = daemonGroup env,
            sandboxAllowNetwork = False,    -- Restrict network access
            sandboxPrivileged = False       -- Run as unprivileged user
        }

    -- Create build directory in store
    buildDir <- createBuildDirectory st env buildId

    -- Setup sandbox with proper privilege
    withSandbox st buildDir sandboxConfig $ \sandboxDir -> do
```

```
        -- Prepare derivation
        let derivationPath = sandboxDir </> "derivation.drv"
        writeDerivationFile st derivationPath derivation

        -- Create builder command
        let builderPath = storePath env </> "libexec/ten-builder"
        let builderArgs = [derivationPath, "--build-id=" ++ show buildId]
        let builderEnv = [
                ("TEN_STORE", storePath env),
                ("TEN_BUILD_DIR", buildDir),
                ("TEN_OUT", buildDir </> "out"),
                ("TEN_TMP", buildDir </> "tmp")
            ]

        -- Create process
        process <- createBuilderProcess st builderPath builderArgs builderEnv

        -- Return process handle
        return process

-- | Drop privileges from root to configured daemon user
dropPrivilegesRoot ::
    DaemonContext 'Daemon ->      -- ^ Daemon context
    DaemonConfig ->               -- ^ Configuration
    TenM 'Build 'Daemon ()
dropPrivilegesRoot context config = TenM $ \sp _ -> do
    -- Only drop privileges if running as root
    uid <- getRealUserID

    when (uid == 0) $ do
        logMessage "Running as root, dropping privileges..."

        -- Get user and group information
        case (daemonUser config, daemonGroup config) of
            (Just userName, Just groupName) -> do
                -- Get user entry
                userEntry <- getUserEntryForName (T.unpack userName)

                -- Get group entry
                groupEntry <- getGroupEntryForName (T.unpack groupName)

                -- Update store permissions before dropping privileges
                ensureStorePermissions (userID userEntry) (groupID groupEntry)

                -- Keep store root owned by root for security
                -- But ensure daemon can read/write store contents

                -- Set supplementary groups first
                setGroups []

                -- Set group ID (must be done before user ID)
                setGroupID (groupID groupEntry)

                -- Set user ID
                setUserID (userID userEntry)

                -- Verify the change
                newUid <- getEffectiveUserID
                when (newUid == 0) $
                    logError "Failed to drop privileges - still running as root!"

                logMessage $ "Successfully dropped privileges to user=" <>
                           userName <> ", group=" <> groupName

            (Just userName, Nothing) -> do
                -- Get user entry
                userEntry <- getUserEntryForName (T.unpack userName)
```

```
                -- Use user's primary group
                let primaryGid = userGroupID userEntry

                -- Set supplementary groups
                setGroups []

                -- Set group ID
                setGroupID primaryGid

                -- Set user ID
                setUserID (userID userEntry)

                logMessage $ "Successfully dropped privileges to user=" <> userName

            _ ->
                logMessage "No user/group specified, continuing to run as root"

-- | Ensure store permissions are appropriate for daemon user
ensureStorePermissions ::
    SPrivilegeTier 'Daemon ->       -- ^ Privilege evidence
    UserID ->                       -- ^ User ID
    GroupID ->                      -- ^ Group ID
    TenM 'Build 'Daemon ()
ensureStorePermissions st uid gid = TenM $ \sp _ -> do
    env <- ask

    -- Get store and state paths
    let storeDir = storePath env
    let tmpDir = tempPath env
    let stateFile = statePath env
    let socketPath = socketPath env

    -- Create necessary directories with proper ownership
    createDirectoryIfMissing True storeDir
    createDirectoryIfMissing True tmpDir
    createDirectoryIfMissing True (takeDirectory stateFile)
    createDirectoryIfMissing True (takeDirectory socketPath)

    -- Create critical store subdirectories
    createDirectoryIfMissing True (storeDir </> "var/ten")
    createDirectoryIfMissing True (storeDir </> "var/ten/db")
    createDirectoryIfMissing True (storeDir </> "var/ten/gcroots")
    createDirectoryIfMissing True (storeDir </> "var/ten/profiles")

    -- Set appropriate ownership and permissions for key directories

    -- Keep store root owned by root, but writable by daemon group
    setFileMode storeDir (ownerReadMode .|. ownerWriteMode .|. ownerExecuteMode .|.
                          groupReadMode .|. groupWriteMode .|. groupExecuteMode)

    -- Set var/ten group writable
    setOwnerAndGroup (storeDir </> "var/ten") 0 gid
    setFileMode (storeDir </> "var/ten") (ownerReadMode .|. ownerWriteMode .|. ownerExecuteMode .|.
                                          groupReadMode .|. groupWriteMode .|. groupExecuteMode)

    -- Set /var/ten/db group writable
    setOwnerAndGroup (storeDir </> "var/ten/db") uid gid
    setFileMode (storeDir </> "var/ten/db") (ownerReadMode .|. ownerWriteMode .|. ownerExecuteMode .|.
                                             groupReadMode .|. groupWriteMode .|. groupExecuteMode)

    -- Set tmpDir to daemon user
    setOwnerAndGroup tmpDir uid gid
    setFileMode tmpDir (ownerReadMode .|. ownerWriteMode .|. ownerExecuteMode .|.
                        groupReadMode .|. groupWriteMode .|. groupExecuteMode)

    -- Ensure socket directory is accessible
```

```
    setFileMode (takeDirectory socketPath) (ownerReadMode .|. ownerWriteMode .|. ownerExecuteMode .|.
                                            groupReadMode .|. groupWriteMode .|. groupExecuteMode .|.
                                            otherReadMode .|. otherExecuteMode)

    logMessage "Set appropriate permissions on store directories"

-- | Run a builder process with dropped privileges
runBuilderProcess ::
    SPrivilegeTier 'Daemon ->       -- ^ Daemon privilege evidence
    FilePath ->                     -- ^ Program path
    [String] ->                     -- ^ Arguments
    [(String, String)] ->           -- ^ Environment
    UserID ->                       -- ^ User ID to run as
    GroupID ->                      -- ^ Group ID to run as
    TenM 'Build 'Daemon (ExitCode, String, String)
runBuilderProcess st program args env uid gid = TenM $ \sp _ -> do
    -- Get current user ID
    currentUid <- getRealUserID

    -- Create process configuration
    let processConfig = (proc program args) {
            env = Just env,
            std_in = NoStream,
            std_out = CreatePipe,
            std_err = CreatePipe
        }

    -- Run the process
    if currentUid == 0
        then do
            -- Running as root, drop privileges to specified UID/GID
            mask $ \restore -> do
                -- Create the process with a pipe
                (_, mbStdout, mbStderr, processHandle) <- createProcess processConfig

                -- Drop privileges for the process using singleton evidence
                dropProcessPrivileges st processHandle uid gid

                -- Read output
                output <- restore $ do
                    stdoutContents <- hGetContents (fromJust mbStdout)
                    stderrContents <- hGetContents (fromJust mbStderr)
                    exitCode <- waitForProcess processHandle
                    return (exitCode, stdoutContents, stderrContents)

                -- Close handles and return
                hClose (fromJust mbStdout)
                hClose (fromJust mbStderr)
                return output
        else do
            -- Not running as root, can't drop privileges
            (exitCode, stdout, stderr) <- readCreateProcessWithExitCode processConfig ""
            return (exitCode, stdout, stderr)

-- | Drop privileges for a specific process
dropProcessPrivileges ::
    SPrivilegeTier 'Daemon ->       -- ^ Daemon privilege evidence
    ProcessHandle ->                -- ^ Process handle
    UserID ->                       -- ^ User ID to run as
    GroupID ->                      -- ^ Group ID to run as
    TenM 'Build 'Daemon ()
dropProcessPrivileges st processHandle uid gid = TenM $ \sp _ -> do
    -- Get process ID from handle
    pid <- getProcessID processHandle

    -- Use system calls to change process credentials
    -- This would use ptrace or similar OS-specific mechanism
```

```
    changeProcessCredentials pid uid gid

    -- Verify the change
    verifyProcessCredentials pid uid
```

2.14 src/Ten/Daemon/State.hs

Ten/Daemon/State.hs Changes

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE RankNTypes #-}

module Ten.Daemon.State where

import Control.Concurrent (MVar, newMVar, takeMVar, putMVar, withMVar)
import Control.Concurrent.STM
import Control.Exception (bracket, finally, try, SomeException)
import Control.Monad (unless, when, void)
import Control.Monad.Reader
import Control.Monad.Except
import Data.Aeson ((.:), (.=))
import qualified Data.Aeson as Aeson
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy as LBS
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T
import Data.Singletons
import System.Directory (doesFileExist, createDirectoryIfMissing, getModificationTime)
import System.IO (Handle, withFile, IOMode(..), hClose)

import Ten.Core
import Ten.Build (BuildResult(..))
import Ten.Derivation (Derivation)

-- | Active build information with privilege awareness
data ActiveBuild (t :: PrivilegeTier) = ActiveBuild {
    abBuildId :: BuildId,
    abDerivation :: Derivation,
    abOwner :: UserId,
    abStatus :: TVar BuildStatus,
    abStartTime :: UTCTime,
    abUpdateTime :: TVar UTCTime,
    abLogBuffer :: TVar Text,
    abResult :: TMVar (Either BuildError BuildResult),
    abProcessId :: TVar (Maybe ProcessID),
    abPrivilegeTier :: PrivilegeTier  -- Record which privilege tier initiated this build
}

-- | Build queue entry
data BuildQueueEntry = BuildQueueEntry {
    bqBuildId :: BuildId,
    bqDerivation :: Derivation,
    bqOwner :: UserId,
    bqPriority :: Int,
    bqPrivilegeTier :: PrivilegeTier  -- Record which privilege tier requested this build
}

-- | Daemon state with privilege tier phantom
```

```haskell
data DaemonState (t :: PrivilegeTier) = DaemonState {
    -- Build tracking (with phantom type for privilege control)
    dsActiveBuilds :: TVar (Map BuildId (ActiveBuild t)),
    dsBuildQueue :: TVar [BuildQueueEntry],
    dsCompletedBuilds :: TVar (Map BuildId (UTCTime, Either BuildError BuildResult)),
    dsFailedBuilds :: TVar (Map BuildId (UTCTime, BuildError)),

    -- Store path tracking
    dsPathLocks :: TVar (Map StorePath (TMVar ())),
    dsReachablePaths :: TVar (Set StorePath),

    -- Derivation tracking
    dsKnownDerivations :: TVar (Map Text Derivation),

    -- GC coordination
    dsGCLock :: TMVar (),
    dsGCLockOwner :: TVar (Maybe ProcessID),

    -- Configuration
    dsStateFilePath :: FilePath,
    dsMaxConcurrentBuilds :: Int,

    -- Privilege tracking
    dsPrivilegeEvidence :: SPrivilegeTier t  -- Runtime evidence of privilege tier
}

-- | Initialize daemon state with appropriate privilege tier
initDaemonState ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    FilePath ->                   -- ^ State file path
    Int ->                        -- ^ Max concurrent builds
    TenM 'Build t (DaemonState t)
initDaemonState st stateFile maxJobs = TenM $ \sp _ -> do
    -- Create TVars for state components
    activeBuildsVar <- newTVarIO Map.empty
    buildQueueVar <- newTVarIO []
    completedBuildsVar <- newTVarIO Map.empty
    failedBuildsVar <- newTVarIO Map.empty

    pathLocksVar <- newTVarIO Map.empty
    reachablePathsVar <- newTVarIO Set.empty

    knownDerivationsVar <- newTVarIO Map.empty

    gcLockVar <- newTMVarIO ()
    gcLockOwnerVar <- newTVarIO Nothing

    -- Return the state with privilege evidence
    return DaemonState {
        dsActiveBuilds = activeBuildsVar,
        dsBuildQueue = buildQueueVar,
        dsCompletedBuilds = completedBuildsVar,
        dsFailedBuilds = failedBuildsVar,

        dsPathLocks = pathLocksVar,
        dsReachablePaths = reachablePathsVar,

        dsKnownDerivations = knownDerivationsVar,

        dsGCLock = gcLockVar,
        dsGCLockOwner = gcLockOwnerVar,

        dsStateFilePath = stateFile,
        dsMaxConcurrentBuilds = maxJobs,

        dsPrivilegeEvidence = st
    }
```

```haskell
-- | Load daemon state from a file with privilege checking
loadStateFromFile ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    FilePath ->                   -- ^ State file path
    Int ->                        -- ^ Max concurrent builds
    TenM 'Build t (DaemonState t)
loadStateFromFile st stateFile maxJobs = TenM $ \sp _ -> do
    -- Check if file exists
    exists <- doesFileExist stateFile

    if not exists
        then do
            -- Create a new state if file doesn't exist
            initDaemonState st stateFile maxJobs
        else do
            -- Try to load from file
            result <- try $ do
                content <- BS.readFile stateFile
                case Aeson.eitherDecodeStrict content of
                    Left err -> throwIO $ StateError $ T.pack err
                    Right stateData -> do
                        -- Initialize empty state
                        state <- initDaemonState st stateFile maxJobs

                        -- Populate with data from file
                        populateState st state stateData

                        return state

            case result of
                Left (err :: SomeException) -> do
                    -- If loading fails, create a fresh state
                    initDaemonState st stateFile maxJobs
                Right state -> return state

-- | Save daemon state to a file
saveStateToFile ::
    SPrivilegeTier 'Daemon ->     -- ^ Daemon privilege evidence
    DaemonState 'Daemon ->       -- ^ Daemon state
    TenM 'Build 'Daemon ()
saveStateToFile st state = TenM $ \sp _ -> do
    -- Create directory if needed
    createDirectoryIfMissing True (takeDirectory (dsStateFilePath state))

    -- Capture state data
    stateData <- captureStateData st state

    -- Write to a temporary file first
    let tempFile = dsStateFilePath state ++ ".tmp"
    BS.writeFile tempFile (LBS.toStrict $ Aeson.encode stateData)

    -- Rename to final location
    renameFile tempFile (dsStateFilePath state)

-- | Register a build with proper privilege tier tracking
registerBuild ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->              -- ^ Daemon state
    Derivation ->                 -- ^ Derivation to build
    UserId ->                     -- ^ Owner
    Int ->                        -- ^ Priority
    TenM 'Build t BuildId
registerBuild st state derivation owner priority = TenM $ \sp _ -> do
    -- Generate a new build ID
    buildId <- genBuildId
```

```
    now <- getCurrentTime

    -- Check if we're at capacity for concurrent builds
    activeBuildCount <- atomically $ Map.size <$> readTVar (dsActiveBuilds state)

    if activeBuildCount >= dsMaxConcurrentBuilds state
        then do
            -- Queue the build
            let queueEntry = BuildQueueEntry {
                    bqBuildId = buildId,
                    bqDerivation = derivation,
                    bqOwner = owner,
                    bqPriority = priority,
                    bqPrivilegeTier = fromSing st  -- Record which tier initiated this build
                }

            atomically $ modifyTVar' (dsBuildQueue state) $ \entries ->
                sortQueueEntries (queueEntry : entries)

            return buildId
        else do
            -- Create a new active build
            statusVar <- newTVarIO BuildPending
            updateTimeVar <- newTVarIO now
            logBufferVar <- newTVarIO ""
            resultVar <- newEmptyTMVarIO
            processIdVar <- newTVarIO Nothing

            let activeBuild = ActiveBuild {
                    abBuildId = buildId,
                    abDerivation = derivation,
                    abOwner = owner,
                    abStatus = statusVar,
                    abStartTime = now,
                    abUpdateTime = updateTimeVar,
                    abLogBuffer = logBufferVar,
                    abResult = resultVar,
                    abProcessId = processIdVar,
                    abPrivilegeTier = fromSing st  -- Record which tier initiated this build
                }

            -- Register the build
            atomically $ modifyTVar' (dsActiveBuilds state) $ Map.insert buildId activeBuild

            -- Register the derivation
            atomically $ modifyTVar' (dsKnownDerivations state) $
                Map.insert (derivHash derivation) derivation

            return buildId

-- | Register a build via protocol (for Builder tier to communicate with Daemon)
registerBuildViaProtocol ::
    SPrivilegeTier 'Builder ->     -- ^ Builder privilege evidence
    DaemonState 'Builder ->        -- ^ Builder state
    Derivation ->                  -- ^ Derivation to build
    UserId ->                      -- ^ Owner
    Int ->                         -- ^ Priority
    TenM 'Build 'Builder BuildId
registerBuildViaProtocol st state derivation owner priority = TenM $ \sp _ -> do
    -- For Builder tier, we need to use the protocol to talk to the daemon
    -- Create a registration request
    let request = RegisterBuildRequest {
            regDerivation = derivation,
            regOwner = owner,
            regPriority = priority
        }
```

```
    -- Send via protocol
    response <- sendToDaemon request

    -- Process response
    case response of
        BuildRegisteredResponse buildId ->
            return buildId
        ErrorResponse err ->
            throwError $ DaemonError $ "Failed to register build: " <> err
        _ ->
            throwError $ DaemonError "Invalid response from daemon for build registration"

-- | Update a build's status with proper privilege checking
updateBuildStatus ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->              -- ^ Daemon state
    BuildId ->                    -- ^ Build ID
    BuildStatus ->                -- ^ New status
    TenM 'Build t ()
updateBuildStatus st state buildId status = TenM $ \sp _ -> do
    -- Check if the build exists
    mBuild <- atomically $ Map.lookup buildId <$> readTVar (dsActiveBuilds state)

    case mBuild of
        Nothing ->
            throwError $ StateBuildNotFound buildId
        Just build -> do
            -- Verify privileges
            case (fromSing st, abPrivilegeTier build) of
                (Builder, Daemon) ->
                    -- Builder tier can't update Daemon-initiated builds
                    throwError $ PrivilegeError "Cannot update daemon-initiated build with builder privileges"
                _ -> do
                    -- Update status
                    now <- getCurrentTime
                    atomically $ do
                        writeTVar (abStatus build) status
                        writeTVar (abUpdateTime build) now

-- | Acquire a store path lock with privilege checking
acquirePathLock ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->              -- ^ Daemon state
    StorePath ->                  -- ^ Path to lock
    TenM 'Build t ()
acquirePathLock st state path = TenM $ \sp _ -> do
    -- Get or create lock for this path
    lock <- atomically $ do
        locks <- readTVar (dsPathLocks state)
        case Map.lookup path locks of
            Just existingLock -> return existingLock
            Nothing -> do
                newLock <- newTMVar ()
                modifyTVar' (dsPathLocks state) $ Map.insert path newLock
                return newLock

    -- Try to acquire the lock
    taken <- atomically $ isEmptyTMVar lock
    unless taken $ do
        atomically $ takeTMVar lock

-- | Release a store path lock
releasePathLock ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->              -- ^ Daemon state
    StorePath ->                  -- ^ Path to unlock
    TenM 'Build t ()
```

```haskell
releasePathLock st state path = TenM $ \sp _ -> do
    -- Get the lock
    mLock <- atomically $ Map.lookup path <$> readTVar (dsPathLocks state)

    case mLock of
        Nothing -> return ()  -- No lock to release
        Just lock -> do
            -- Check if lock is taken
            empty <- atomically $ isEmptyTMVar lock

            unless empty $ do
                -- Release the lock
                atomically $ putTMVar lock ()

-- | Execute an action with a locked store path
withPathLock ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->             -- ^ Daemon state
    StorePath ->                  -- ^ Path to lock
    TenM 'Build t a ->            -- ^ Action to run with lock
    TenM 'Build t a
withPathLock st state path action = TenM $ \sp _ -> do
    bracket
        (runTenM (acquirePathLock st state path) sp st)
        (\_ -> runTenM (releasePathLock st state path) sp st)
        (\_ -> runTenM action sp st)

-- | Acquire the GC lock with proper privilege verification
acquireGCLock ::
    SPrivilegeTier 'Daemon ->      -- ^ Daemon privilege evidence
    DaemonState 'Daemon ->        -- ^ Daemon state
    TenM 'Build 'Daemon ()
acquireGCLock st state = TenM $ \sp _ -> do
    -- Verify we're in daemon context (this is enforced by type system too)
    let privilegeTier = fromSing st
    when (privilegeTier /= Daemon) $
        throwError $ PrivilegeError "GC operations require daemon privileges"

    -- Try to take the TMVar lock
    empty <- atomically $ isEmptyTMVar (dsGCLock state)
    unless empty $ do
        atomically $ takeTMVar (dsGCLock state)

    -- Record our process as the owner
    pid <- getProcessID
    atomically $ writeTVar (dsGCLockOwner state) (Just pid)

-- | Release the GC lock
releaseGCLock ::
    SPrivilegeTier 'Daemon ->      -- ^ Daemon privilege evidence
    DaemonState 'Daemon ->        -- ^ Daemon state
    TenM 'Build 'Daemon ()
releaseGCLock st state = TenM $ \sp _ -> do
    -- Clear the owner
    atomically $ writeTVar (dsGCLockOwner state) Nothing

    -- Release the lock
    empty <- atomically $ isEmptyTMVar (dsGCLock state)
    when empty $ do
        atomically $ putTMVar (dsGCLock state) ()

-- | Execute an action with the GC lock
withGCLock ::
    SPrivilegeTier 'Daemon ->      -- ^ Daemon privilege evidence
    DaemonState 'Daemon ->        -- ^ Daemon state
    (SPrivilegeTier 'Daemon -> TenM 'Build 'Daemon a) -> -- ^ Action to run with lock
    TenM 'Build 'Daemon a
```

```haskell
withGCLock st state action = TenM $ \sp _ -> do
    bracket
        (runTenM (acquireGCLock st state) sp st)
        (\_ -> runTenM (releaseGCLock st state) sp st)
        (\_ -> runTenM (action st) sp st)

-- | Check if GC lock is available
checkGCLock ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->              -- ^ Daemon state
    TenM 'Build t Bool
checkGCLock st state = TenM $ \sp _ -> do
    -- Check if we have a recorded owner
    mOwner <- atomically $ readTVar (dsGCLockOwner state)
    return $ isNothing mOwner

-- | Mark a path as reachable
markPathAsReachable ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->              -- ^ Daemon state
    StorePath ->                  -- ^ Path to mark
    TenM 'Build t ()
markPathAsReachable st state path = TenM $ \sp _ -> do
    atomically $ modifyTVar' (dsReachablePaths state) $ Set.insert path

-- | Check if a path is reachable
isPathReachable ::
    SPrivilegeTier t ->           -- ^ Privilege evidence
    DaemonState t ->              -- ^ Daemon state
    StorePath ->                  -- ^ Path to check
    TenM 'Build t Bool
isPathReachable st state path = TenM $ \sp _ -> do
    atomically $ Set.member path <$> readTVar (dsReachablePaths state)

-- | Run background maintenance tasks
scheduledMaintenance ::
    SPrivilegeTier 'Daemon ->     -- ^ Daemon privilege evidence
    DaemonState 'Daemon ->        -- ^ Daemon state
    TenM 'Build 'Daemon ()
scheduledMaintenance st state = TenM $ \sp _ -> do
    -- Prune old completed builds
    pruneCompletedBuilds st state

    -- Clean up stale builds
    cleanupStaleBuilds st state

    -- Save state to file
    saveStateToFile st state

-- | The Builder-safe view of the daemon state for RPC
-- This allows builder-privileged code to view/modify only authorized parts
builderViewOfState ::
    SPrivilegeTier 'Builder ->    -- ^ Builder privilege evidence
    DaemonState 'Daemon ->        -- ^ Daemon state (still has Daemon phantom type)
    DaemonState 'Builder          -- ^ Builder view (has Builder phantom type)
builderViewOfState st daemonState =
    -- This would be implemented with a proper restricted view
    -- of the daemon state that only exposes builder-safe operations
    -- Without the full state, privileged operations will fail to compile
    error "Not implemented: converting daemon state to builder view"
```

2.15 src/Ten/Daemon/Auth.hs

Ten/Daemon/Auth.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RankNTypes #-}

module Ten.Daemon.Auth where

import Control.Concurrent.MVar (MVar, newMVar, withMVar)
import Control.Exception (catch, throwIO, Exception)
import Control.Monad (unless, when)
import Control.Monad.Reader
import Control.Monad.Except
import qualified Crypto.Hash as Crypto
import qualified Crypto.KDF.PBKDF2 as PBKDF2
import qualified Data.ByteString as BS
import qualified Data.ByteString.Base64 as Base64
import qualified Data.ByteArray as BA
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import Data.Singletons
import Data.Time.Clock (UTCTime, getCurrentTime, diffUTCTime)
import System.Directory (doesFileExist, createDirectoryIfMissing, renameFile)
import System.FilePath (takeDirectory)
import System.IO (Handle, withFile, IOMode(..), hClose)
import System.Posix.User (UserEntry(..), getUserEntryForName)

import Ten.Core

-- | Authentication error with privilege information
data AuthError
    = InvalidCredentials
    | UserNotFound
    | TokenExpired
    | InsufficientPrivileges PrivilegeTier
    | AuthFileError Text
    deriving (Show, Eq)

instance Exception AuthError

-- | User credentials for authentication
data UserCredentials = UserCredentials {
    ucUsername :: Text,
    ucPassword :: Text,
    ucRequestedTier :: PrivilegeTier  -- Requested privilege tier
} deriving (Show, Eq)

-- | Token information with privilege tier
data TokenInfo = TokenInfo {
    tiToken :: Text,
    tiUserId :: UserId,
    tiExpires :: Maybe UTCTime,
    tiPermissions :: Set Permission,
    tiPrivilegeTier :: PrivilegeTier,  -- Granted privilege tier
    tiLastUsed :: UTCTime
} deriving (Show, Eq)

-- | Password hash
```

```haskell
data PasswordHash = PasswordHash {
    phAlgorithm :: Text,
    phSalt :: BS.ByteString,
    phHash :: BS.ByteString,
    phIterations :: Int
} deriving (Show, Eq)

-- | User information with permissions and privilege tiers
data UserInfo = UserInfo {
    uiUserId :: UserId,
    uiUsername :: Text,
    uiPasswordHash :: PasswordHash,
    uiPermissions :: Set Permission,
    uiAllowedPrivilegeTiers :: Set PrivilegeTier, -- Which privilege tiers are allowed
    uiTokens :: Map Text TokenInfo,
    uiLastLogin :: Maybe UTCTime,
    uiSystemUser :: Maybe Text  -- Associated system user, if any
} deriving (Show, Eq)

-- | Authentication database
data AuthDb = AuthDb {
    adUsers :: Map Text UserInfo,
    adTokens :: Map Text Text,    -- Token to username mapping
    adLastModified :: UTCTime
} deriving (Show, Eq)

-- | Permission types with associated privilege requirements
data Permission
    = PermBuild              -- Can build derivations (Builder)
    | PermCancelBuild        -- Can cancel builds (Builder)
    | PermQueryBuild         -- Can query build status (Builder)
    | PermQueryStore         -- Can query store contents (Builder)
    | PermModifyStore        -- Can add to store (Daemon)
    | PermRunGC              -- Can run garbage collection (Daemon)
    | PermShutdown           -- Can shut down daemon (Daemon)
    | PermManageUsers        -- Can manage users (Daemon)
    | PermAdmin              -- Administrative access (Daemon)
    deriving (Show, Eq, Ord, Enum, Bounded)

-- | Check if a permission requires daemon privilege
permissionRequiresDaemon :: Permission -> Bool
permissionRequiresDaemon PermModifyStore = True
permissionRequiresDaemon PermRunGC = True
permissionRequiresDaemon PermShutdown = True
permissionRequiresDaemon PermManageUsers = True
permissionRequiresDaemon PermAdmin = True
permissionRequiresDaemon _ = False

-- | Type families for permission capabilities
type family CanModifyStore (t :: PrivilegeTier) :: Bool where
    CanModifyStore 'Daemon = 'True
    CanModifyStore 'Builder = 'False

type family CanRunGC (t :: PrivilegeTier) :: Bool where
    CanRunGC 'Daemon = 'True
    CanRunGC 'Builder = 'False

type family CanManageUsers (t :: PrivilegeTier) :: Bool where
    CanManageUsers 'Daemon = 'True
    CanManageUsers 'Builder = 'False

-- | Default permissions for privilege tiers
defaultPermissions :: PrivilegeTier -> Set Permission
defaultPermissions Daemon = Set.fromList [
    PermBuild, PermCancelBuild, PermQueryBuild, PermQueryStore,
    PermModifyStore, PermRunGC, PermShutdown, PermManageUsers, PermAdmin
    ]
```

```haskell
defaultPermissions Builder = Set.fromList [
    PermBuild, PermCancelBuild, PermQueryBuild, PermQueryStore
    ]

-- | Authenticate a user and return proper privilege evidence
authenticateUser ::
    SPrivilegeTier 'Daemon ->       -- ^ Daemon privilege evidence
    Text ->                         -- ^ Username
    Text ->                         -- ^ Password
    PrivilegeTier ->                -- ^ Requested privilege tier
    TenM 'Build 'Daemon (Either Text (UserId, AuthToken, Set Permission, SPrivilegeTier t))
authenticateUser st username password requestedTier = TenM $ \sp _ -> do
    -- Load auth database
    authDb <- loadAuthDb

    -- Find user
    case Map.lookup username (adUsers authDb) of
        Nothing ->
            return $ Left "User not found"

        Just userInfo -> do
            -- Verify password
            passwordValid <- verifyPassword password (uiPasswordHash userInfo)

            if not passwordValid
                then return $ Left "Invalid credentials"
                else do
                    -- Check if the requested privilege tier is allowed for this user
                    let allowedTiers = uiAllowedPrivilegeTiers userInfo

                    if requestedTier `Set.member` allowedTiers
                        then do
                            -- Generate token
                            now <- getCurrentTime
                            token <- generateToken

                            -- Grant appropriate permissions based on tier
                            let grantedPermissions =
                                    if requestedTier == Daemon
                                    then uiPermissions userInfo  -- All permissions for Daemon
                                    else Set.filter (not . permissionRequiresDaemon) (uiPermissions userInfo)

                            -- Create token info
                            let tokenInfo = TokenInfo {
                                    tiToken = token,
                                    tiUserId = uiUserId userInfo,
                                    tiExpires = Just $ addUTCTime (60*60*24) now, -- 24h expiry
                                    tiPermissions = grantedPermissions,
                                    tiPrivilegeTier = requestedTier,
                                    tiLastUsed = now
                                }

                            -- Update auth database
                            let updatedTokens = Map.insert token tokenInfo (uiTokens userInfo)
                            let updatedUser = userInfo {
                                    uiTokens = updatedTokens,
                                    uiLastLogin = Just now
                                }
                            let updatedUsers = Map.insert username updatedUser (adUsers authDb)
                            let updatedTokenMap = Map.insert token username (adTokens authDb)
                            let updatedDb = authDb {
                                    adUsers = updatedUsers,
                                    adTokens = updatedTokenMap,
                                    adLastModified = now
                                }

                            -- Save updated auth database
```

```
                            saveAuthDb updatedDb

                            -- Return appropriate privilege evidence
                            case requestedTier of
                                Daemon ->
                                    -- Return daemon privilege evidence
                                    return $ Right (uiUserId userInfo, AuthToken token, grantedPermissions,
                                                SDaemon)

                                Builder ->
                                    -- Return builder privilege evidence
                                    return $ Right (uiUserId userInfo, AuthToken token, grantedPermissions,
                                                SBuilder)
                        else
                            return $ Left $ "User not authorized for " <> T.pack (show requestedTier) <> "
                            ↪  privileges"

-- | Validate a token with proper privilege evidence
validateToken ::
    SPrivilegeTier 'Daemon ->              -- ^ Daemon privilege evidence
    AuthToken ->                            -- ^ Authentication token
    TenM 'Build 'Daemon (Either Text (UserId, Set Permission, SPrivilegeTier t))
validateToken st (AuthToken token) = TenM $ \sp _ -> do
    -- Load auth database
    authDb <- loadAuthDb

    -- Find the token
    case Map.lookup token (adTokens authDb) of
        Nothing ->
            return $ Left "Invalid or expired token"

        Just username -> do
            -- Find the user
            case Map.lookup username (adUsers authDb) of
                Nothing ->
                    return $ Left "User not found"

                Just userInfo -> do
                    -- Find token info
                    case Map.lookup token (uiTokens userInfo) of
                        Nothing ->
                            return $ Left "Token not found"

                        Just tokenInfo -> do
                            -- Check expiration
                            now <- getCurrentTime
                            case tiExpires tokenInfo of
                                Just expiry | now > expiry ->
                                    return $ Left "Token expired"
                                _ -> do
                                    -- Update last used time
                                    let updatedTokenInfo = tokenInfo { tiLastUsed = now }
                                    let updatedTokens = Map.insert token updatedTokenInfo (uiTokens userInfo)
                                    let updatedUser = userInfo { uiTokens = updatedTokens }
                                    let updatedUsers = Map.insert username updatedUser (adUsers authDb)
                                    let updatedDb = authDb {
                                            adUsers = updatedUsers,
                                            adLastModified = now
                                        }

                                    -- Save updated auth database
                                    saveAuthDb updatedDb

                                    -- Return appropriate singleton evidence based on token's tier
                                    case tiPrivilegeTier tokenInfo of
                                        Daemon ->
                                            return $ Right (tiUserId tokenInfo, tiPermissions tokenInfo, SDaemon)
```

```
                                        Builder ->
                                            return $ Right (tiUserId tokenInfo, tiPermissions tokenInfo,
                                            ↪  SBuilder)

-- | Verify if a password matches a hash
verifyPassword ::
    Text ->                  -- ^ Password to check
    PasswordHash ->          -- ^ Password hash
    TenM 'Build t Bool
verifyPassword password PasswordHash{..} = TenM $ \_ _ -> do
    case phAlgorithm of
        "pbkdf2-sha512" ->
            let calculatedHash = pbkdf2Hash (TE.encodeUtf8 password) phSalt phIterations 64
            in return $ calculatedHash == phHash
        _ -> return False  -- Unknown algorithm

-- | Generate PBKDF2 hash
pbkdf2Hash :: BS.ByteString -> BS.ByteString -> Int -> Int -> BS.ByteString
pbkdf2Hash password salt iterations keyLen =
    PBKDF2.fastPBKDF2_SHA512
        (PBKDF2.Parameters iterations keyLen)
        password
        salt

-- | Hash a password
hashPassword ::
    SPrivilegeTier 'Daemon ->   -- ^ Privilege evidence
    Text ->                     -- ^ Password to hash
    TenM 'Build 'Daemon PasswordHash
hashPassword st password = TenM $ \_ _ -> do
    -- Generate a random salt (16 bytes)
    salt <- generateRandomBytes 16

    -- Use 100,000 iterations (adjust based on security requirements)
    let iterations = 100000
    let hash = pbkdf2Hash (TE.encodeUtf8 password) salt iterations 64

    return PasswordHash {
        phAlgorithm = "pbkdf2-sha512",
        phSalt = salt,
        phHash = hash,
        phIterations = iterations
    }

-- | Generate a random authentication token
generateToken :: TenM 'Build t Text
generateToken = TenM $ \_ _ -> do
    -- Generate random bytes (32 bytes = 256 bits of entropy)
    randomBytes <- generateRandomBytes 32

    -- Use Base64 for text representation
    let token = "ten_" <> TE.decodeUtf8 (Base64.encode randomBytes)
    return token

-- | Add a new user with proper privilege checks
addUser ::
    SPrivilegeTier 'Daemon ->       -- ^ Daemon privilege evidence
    Text ->                         -- ^ Username
    Text ->                         -- ^ Password
    Set Permission ->               -- ^ Permissions
    Set PrivilegeTier ->            -- ^ Allowed privilege tiers
    TenM 'Build 'Daemon ()
addUser st username password permissions allowedTiers = TenM $ \sp _ -> do
    -- Verify we have admin privileges (already checked by type system)

    -- Load auth database
```

```
    authDb <- loadAuthDb

    -- Check if user already exists
    when (Map.member username (adUsers authDb)) $
        throwError $ AuthError $ "User already exists: " <> username

    -- Hash the password
    passwordHash <- hashPassword st password

    -- Generate user ID
    userId <- UserId <$> ("user_" <>) <$> uniqueId

    now <- getCurrentTime

    -- Create user info
    let userInfo = UserInfo {
            uiUserId = userId,
            uiUsername = username,
            uiPasswordHash = passwordHash,
            uiPermissions = permissions,
            uiAllowedPrivilegeTiers = allowedTiers,
            uiTokens = Map.empty,
            uiLastLogin = Nothing,
            uiSystemUser = Nothing
        }

    -- Update auth database
    let updatedUsers = Map.insert username userInfo (adUsers authDb)
    let updatedDb = authDb {
            adUsers = updatedUsers,
            adLastModified = now
        }

    -- Save auth database
    saveAuthDb updatedDb

-- | Remove a user with proper privilege checks
removeUser ::
    SPrivilegeTier 'Daemon ->       -- ^ Daemon privilege evidence
    Text ->                         -- ^ Username
    TenM 'Build 'Daemon ()
removeUser st username = TenM $ \sp _ -> do
    -- Load auth database
    authDb <- loadAuthDb

    -- Check if user exists
    case Map.lookup username (adUsers authDb) of
        Nothing ->
            throwError $ AuthError $ "User not found: " <> username

        Just userInfo -> do
            now <- getCurrentTime

            -- Remove all tokens for this user
            let userTokens = Map.keys (uiTokens userInfo)
            let updatedTokenMap = foldl (\m t -> Map.delete t m) (adTokens authDb) userTokens

            -- Update auth database
            let updatedUsers = Map.delete username (adUsers authDb)
            let updatedDb = authDb {
                    adUsers = updatedUsers,
                    adTokens = updatedTokenMap,
                    adLastModified = now
                }

            -- Save auth database
            saveAuthDb updatedDb
```

```haskell
-- | Change a user's allowed privilege tiers
changeUserPrivilegeTiers ::
    SPrivilegeTier 'Daemon ->       -- ^ Daemon privilege evidence
    Text ->                         -- ^ Username
    Set PrivilegeTier ->            -- ^ New allowed tiers
    TenM 'Build 'Daemon ()
changeUserPrivilegeTiers st username newTiers = TenM $ \sp _ -> do
    -- Load auth database
    authDb <- loadAuthDb

    -- Check if user exists
    case Map.lookup username (adUsers authDb) of
        Nothing ->
            throwError $ AuthError $ "User not found: " <> username

        Just userInfo -> do
            now <- getCurrentTime

            -- Update user info
            let updatedUser = userInfo { uiAllowedPrivilegeTiers = newTiers }

            -- Update auth database
            let updatedUsers = Map.insert username updatedUser (adUsers authDb)
            let updatedDb = authDb {
                    adUsers = updatedUsers,
                    adLastModified = now
                }

            -- Save auth database
            saveAuthDb updatedDb

-- | Check if a user can use a specific permission in the current context
checkPermission ::
    SPrivilegeTier t ->             -- ^ Privilege evidence
    Set Permission ->               -- ^ User permissions
    Permission ->                   -- ^ Permission to check
    TenM 'Build t Bool
checkPermission st permissions perm = TenM $ \sp _ -> do
    -- First check if the permission is in the set
    let hasPermission = perm `Set.member` permissions

    -- If not in the set, fail immediately
    if not hasPermission
        then return False
        else do
            -- If permission requires daemon privileges, check the privilege tier
            case permissionRequiresDaemon perm of
                True ->
                    -- For daemon privileges, check singleton type
                    case fromSing st of
                        Daemon -> return True   -- Has daemon privileges
                        Builder -> return False  -- Doesn't have daemon privileges
                False ->
                    -- Permission doesn't require daemon privileges
                    return True

-- | Load the authentication database
loadAuthDb :: TenM 'Build t AuthDb
loadAuthDb = TenM $ \_ _ -> do
    -- Get auth database path
    dbPath <- getAuthDbPath

    -- Check if file exists
    exists <- doesFileExist dbPath

    if not exists
```

```
        then do
            -- Create new empty database
            now <- getCurrentTime
            return AuthDb {
                adUsers = Map.empty,
                adTokens = Map.empty,
                adLastModified = now
            }
        else do
            -- Read the file
            content <- BS.readFile dbPath

            -- Parse the database
            case Aeson.eitherDecodeStrict content of
                Left err ->
                    throwIO $ AuthError $ AuthFileError $ "Failed to parse auth database: " <> T.pack err

                Right db -> return db

-- | Save the authentication database
saveAuthDb :: AuthDb -> TenM 'Build t ()
saveAuthDb db = TenM $ \_ _ -> do
    -- Get auth database path
    dbPath <- getAuthDbPath

    -- Create directory if needed
    createDirectoryIfMissing True (takeDirectory dbPath)

    -- Write to a temporary file first
    let tempPath = dbPath <> ".tmp"
    BS.writeFile tempPath (LBS.toStrict $ Aeson.encode db)

    -- Rename to final path
    renameFile tempPath dbPath

-- | Get auth database path
getAuthDbPath :: IO FilePath
getAuthDbPath = do
    -- Get home directory
    homeDir <- getHomeDirectory

    -- Use XDG directories if available
    mXdgConfigDir <- lookupEnv "XDG_CONFIG_HOME"

    case mXdgConfigDir of
        Just configDir ->
            return $ configDir </> "ten/auth.db"
        Nothing ->
            return $ homeDir </> ".config/ten/auth.db"

-- | Generate random bytes
generateRandomBytes :: Int -> IO BS.ByteString
generateRandomBytes n = do
    -- In a real implementation, use a CSPRNG
    BS.pack <$> replicateM n (randomRIO (0, 255))

-- | Generate a unique ID string
uniqueId :: IO Text
uniqueId = do
    -- Get current time
    now <- getCurrentTime

    -- Convert to microseconds since epoch
    let micros = floor $ 1000000 * realToFrac (diffUTCTime now (read "1970-01-01 00:00:00 UTC"))

    -- Add some randomness
    r <- randomIO
```

```haskell
    -- Combine time and random number
    return $ T.pack $ show micros <> "_" <> show (r `mod` 1000)
```

2.16 src/Ten/Daemon/Config.hs

Ten/Daemon/Config.hs Changes

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Ten.Daemon.Config where

import Control.Exception (try, SomeException)
import Control.Monad (when, unless)
import Data.Aeson ((.:), (.=))
import qualified Data.Aeson as Aeson
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy as LBS
import Data.Maybe (fromMaybe, isNothing)
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Text (Text)
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import System.Console.GetOpt (OptDescr(..), ArgDescr(..), ArgOrder(..), getOpt)
import System.Directory (doesFileExist, getHomeDirectory, createDirectoryIfMissing)
import System.Environment (getEnv, lookupEnv)
import System.FilePath ((</>), takeDirectory)
import System.Posix.User (getEffectiveUserID, getUserEntryForID, userName)

import Ten.Core

-- | Daemon configuration with privilege-aware settings
data DaemonConfig = DaemonConfig {
    -- Network settings
    daemonSocketPath    :: FilePath,      -- ^ Path to the daemon socket
    daemonPort          :: Maybe Int,     -- ^ Optional TCP port for remote connections
    daemonBindAddress   :: Maybe String,  -- ^ Optional bind address for TCP connections

    -- Storage settings
    daemonStorePath     :: FilePath,      -- ^ Path to the content-addressable store
    daemonStateFile     :: FilePath,      -- ^ Path to the daemon state file
    daemonTmpDir        :: FilePath,      -- ^ Path for temporary files

    -- Build settings
    daemonMaxJobs       :: Int,           -- ^ Maximum concurrent build jobs
    daemonBuildTimeout  :: Int,           -- ^ Build timeout in seconds (0 = no timeout)
    daemonKeepFailed    :: Bool,          -- ^ Keep failed build outputs for inspection

    -- Privilege and security settings
    daemonUser          :: Maybe Text,    -- ^ User to run daemon as (if running as root)
    daemonGroup         :: Maybe Text,    -- ^ Group to run daemon as (if running as root)
    daemonAllowedUsers  :: Set Text,      -- ^ Users allowed to connect to daemon
    daemonRequireAuth   :: Bool,          -- ^ Whether authentication is required
    daemonAllowPrivilegeEscalation :: Bool, -- ^ Whether to allow privilege escalation
    daemonPrivilegeModel :: PrivilegeModel, -- ^ How to handle privileges

    -- Logging settings
    daemonLogFile       :: Maybe FilePath, -- ^ Path to log file (Nothing = stdout)
    daemonLogLevel      :: LogLevel,       -- ^ Logging verbosity

    -- Daemon behavior settings
    daemonForeground    :: Bool,          -- ^ Run in foreground (don't daemonize)
    daemonAutoStart     :: Bool,          -- ^ Auto-start daemon when client connects if not running
```

```
    daemonRestrictive   :: Bool           -- ^ Use more restrictive sandbox settings
}

-- | Log levels
data LogLevel
    = LogQuiet      -- ^ Minimal logging
    | LogNormal     -- ^ Standard logging
    | LogVerbose    -- ^ Verbose logging
    | LogDebug      -- ^ Debug level logging
    deriving (Show, Eq, Ord, Enum, Bounded)

-- | Privilege handling model
data PrivilegeModel
    = PrivilegeAlwaysDrop     -- ^ Always drop privileges (safest)
    | PrivilegeDropSelective  -- ^ Drop privileges selectively
    | PrivilegeNoDrop         -- ^ Never drop privileges (only for special cases)
    deriving (Show, Eq)

-- | Load daemon configuration from a file
loadDaemonConfig ::
    SPrivilegeTier t ->       -- ^ Privilege evidence
    FilePath ->               -- ^ Config file path
    TenM 'Build t (Either String DaemonConfig)
loadDaemonConfig st path = TenM $ \_ _ -> do
    -- Check if the file exists
    exists <- doesFileExist path

    if not exists
        then return $ Left $ "Configuration file not found: " ++ path
        else do
            -- Read and parse the file
            result <- try $ BS.readFile path
            case result of
                Left (ex :: SomeException) ->
                    return $ Left $ "Error reading configuration file: " ++ show ex

                Right content ->
                    case Aeson.eitherDecodeStrict content of
                        Left err ->
                            return $ Left $ "Error parsing configuration: " ++ err

                        Right config -> do
                            -- Validate the loaded configuration
                            case validateConfig config of
                                Left err -> return $ Left err
                                Right validConfig -> return $ Right validConfig

-- | Get default configuration considering user permissions
getDefaultConfig :: TenM 'Build t DaemonConfig
getDefaultConfig = TenM $ \_ _ -> do
    -- Get current user information
    uid <- getEffectiveUserID
    userEntry <- getUserEntryForID uid
    let currentUser = T.pack $ userName userEntry

    -- Determine if we're running as root
    let isRoot = uid == 0

    -- Get home directory for non-root paths
    homeDir <- getHomeDirectory

    -- Get XDG directories
    xdgConfigHome <- lookupEnv "XDG_CONFIG_HOME"
    let xdgConfigDir = fromMaybe (homeDir </> ".config") xdgConfigHome
    let tenConfigDir = xdgConfigDir </> "ten"

    xdgDataHome <- lookupEnv "XDG_DATA_HOME"
```

```
    let xdgDataDir = fromMaybe (homeDir </> ".local/share") xdgDataHome
    let tenDataDir = xdgDataDir </> "ten"

    xdgStateHome <- lookupEnv "XDG_STATE_HOME"
    let xdgStateDir = fromMaybe (homeDir </> ".local/state") xdgStateHome
    let tenStateDir = xdgStateDir </> "ten"

    -- Create directories if they don't exist
    mapM_ (createDirectoryIfMissing True)
          [tenConfigDir, tenDataDir, tenStateDir]

    -- Modify default config based on whether we're root or not
    if isRoot
        then return defaultDaemonConfig
        else return defaultDaemonConfig {
            -- Non-root appropriate paths
            daemonSocketPath    = tenStateDir </> "daemon.sock",
            daemonStorePath     = tenDataDir </> "store",
            daemonStateFile     = tenStateDir </> "daemon-state.json",
            daemonTmpDir        = tenStateDir </> "tmp",
            daemonLogFile       = Just (tenStateDir </> "daemon.log"),

            -- Always run in foreground when not root
            daemonForeground    = True,

            -- Set current user as default allowed user
            daemonAllowedUsers  = Set.singleton currentUser,

            -- Non-root users can't allow privilege escalation
            daemonAllowPrivilegeEscalation = False
        }

-- | Default daemon configuration
defaultDaemonConfig :: DaemonConfig
defaultDaemonConfig = DaemonConfig {
    -- Network defaults
    daemonSocketPath    = "/var/run/ten/daemon.sock",
    daemonPort          = Nothing,
    daemonBindAddress   = Nothing,

    -- Storage defaults
    daemonStorePath     = "/var/lib/ten/store",
    daemonStateFile     = "/var/lib/ten/daemon-state.json",
    daemonTmpDir        = "/var/lib/ten/tmp",

    -- Build defaults
    daemonMaxJobs       = 4,
    daemonBuildTimeout  = 3600,  -- 1 hour
    daemonKeepFailed    = False,

    -- Privilege and security defaults
    daemonUser          = Nothing,
    daemonGroup         = Nothing,
    daemonAllowedUsers  = Set.empty,  -- Empty means all users allowed
    daemonRequireAuth   = True,
    daemonAllowPrivilegeEscalation = False,  -- By default, don't allow privilege escalation
    daemonPrivilegeModel = PrivilegeAlwaysDrop,  -- Most secure default

    -- Logging defaults
    daemonLogFile       = Just "/var/log/ten/daemon.log",
    daemonLogLevel      = LogNormal,

    -- Behavior defaults
    daemonForeground    = False,
    daemonAutoStart     = True,
    daemonRestrictive   = False
}
```

```haskell
-- | Save configuration to a file
saveConfigToFile ::
    SPrivilegeTier 'Daemon ->      -- ^ Daemon privilege evidence
    FilePath ->                    -- ^ Config file path
    DaemonConfig ->                -- ^ Configuration to save
    TenM 'Build 'Daemon (Either String ())
saveConfigToFile st path config = TenM $ \_ _ -> do
    -- Create directory if needed
    let dir = takeDirectory path
    result <- try $ createDirectoryIfMissing True dir

    case result of
        Left (ex :: SomeException) ->
            return $ Left $ "Error creating directory: " ++ show ex

        Right () -> do
            -- Write the config
            writeResult <- try $ LBS.writeFile path (Aeson.encode config)

            case writeResult of
                Left (ex :: SomeException) ->
                    return $ Left $ "Error writing configuration: " ++ show ex

                Right () ->
                    return $ Right ()

-- | Load configuration from environment variables
loadConfigFromEnv :: TenM 'Build t DaemonConfig
loadConfigFromEnv = TenM $ \_ _ -> do
    -- Start with default config
    defaultConfig <- getDefaultConfig

    -- Override with environment variables
    mSocketPath <- lookupEnv "TEN_DAEMON_SOCKET"
    mPort <- lookupEnvInt "TEN_DAEMON_PORT"
    mBindAddr <- lookupEnv "TEN_DAEMON_BIND_ADDRESS"
    mStorePath <- lookupEnv "TEN_STORE_PATH"
    mStateFile <- lookupEnv "TEN_DAEMON_STATE_FILE"
    mTmpDir <- lookupEnv "TEN_DAEMON_TMP_DIR"
    mMaxJobs <- lookupEnvInt "TEN_DAEMON_MAX_JOBS"
    mTimeout <- lookupEnvInt "TEN_DAEMON_BUILD_TIMEOUT"
    mKeepFailed <- lookupEnvBool "TEN_DAEMON_KEEP_FAILED"
    mUser <- lookupEnvText "TEN_DAEMON_USER"
    mGroup <- lookupEnvText "TEN_DAEMON_GROUP"
    mAllowedUsers <- lookupEnvTextList "TEN_DAEMON_ALLOWED_USERS"
    mRequireAuth <- lookupEnvBool "TEN_DAEMON_REQUIRE_AUTH"
    mAllowPrivEsc <- lookupEnvBool "TEN_DAEMON_ALLOW_PRIVILEGE_ESCALATION"
    mPrivModel <- lookupEnvPrivModel "TEN_DAEMON_PRIVILEGE_MODEL"
    mLogFile <- lookupEnv "TEN_DAEMON_LOG_FILE"
    mLogLevel <- lookupEnvLogLevel "TEN_DAEMON_LOG_LEVEL"
    mForeground <- lookupEnvBool "TEN_DAEMON_FOREGROUND"
    mAutoStart <- lookupEnvBool "TEN_DAEMON_AUTO_START"
    mRestrictive <- lookupEnvBool "TEN_DAEMON_RESTRICTIVE"

    -- Apply all overrides
    let config = defaultConfig {
            daemonSocketPath = fromMaybe (daemonSocketPath defaultConfig) mSocketPath,
            daemonPort = mPort <|> daemonPort defaultConfig,
            daemonBindAddress = mBindAddr <|> daemonBindAddress defaultConfig,
            daemonStorePath = fromMaybe (daemonStorePath defaultConfig) mStorePath,
            daemonStateFile = fromMaybe (daemonStateFile defaultConfig) mStateFile,
            daemonTmpDir = fromMaybe (daemonTmpDir defaultConfig) mTmpDir,
            daemonMaxJobs = fromMaybe (daemonMaxJobs defaultConfig) mMaxJobs,
            daemonBuildTimeout = fromMaybe (daemonBuildTimeout defaultConfig) mTimeout,
            daemonKeepFailed = fromMaybe (daemonKeepFailed defaultConfig) mKeepFailed,
            daemonUser = mUser <|> daemonUser defaultConfig,
```

```
            daemonGroup = mGroup <|> daemonGroup defaultConfig,
            daemonAllowedUsers = maybe (daemonAllowedUsers defaultConfig) Set.fromList mAllowedUsers,
            daemonRequireAuth = fromMaybe (daemonRequireAuth defaultConfig) mRequireAuth,
            daemonAllowPrivilegeEscalation = fromMaybe (daemonAllowPrivilegeEscalation defaultConfig)
            ↪  mAllowPrivEsc,
            daemonPrivilegeModel = fromMaybe (daemonPrivilegeModel defaultConfig) mPrivModel,
            daemonLogFile =
                if mLogFile == Just "stdout"
                then Nothing
                else mLogFile <|> daemonLogFile defaultConfig,
            daemonLogLevel = fromMaybe (daemonLogLevel defaultConfig) mLogLevel,
            daemonForeground = fromMaybe (daemonForeground defaultConfig) mForeground,
            daemonAutoStart = fromMaybe (daemonAutoStart defaultConfig) mAutoStart,
            daemonRestrictive = fromMaybe (daemonRestrictive defaultConfig) mRestrictive
        }

    -- Validate the final config
    case validateConfig config of
        Left err -> error $ "Invalid configuration from environment: " ++ err
        Right validConfig -> return validConfig

-- | Command-line option descriptions for daemon configuration
configOptionDescriptions :: [OptDescr (DaemonConfig -> DaemonConfig)]
configOptionDescriptions = [
    Option ['s'] ["socket"] (ReqArg (\s cfg -> cfg { daemonSocketPath = s }) "PATH")
        "Path to daemon socket",
    Option ['p'] ["port"] (ReqArg (\p cfg -> cfg { daemonPort = Just (read p) }) "PORT")
        "TCP port for remote connections",
    Option ['b'] ["bind"] (ReqArg (\b cfg -> cfg { daemonBindAddress = Just b }) "ADDRESS")
        "Bind address for TCP connections",
    Option [] ["store"] (ReqArg (\s cfg -> cfg { daemonStorePath = s }) "PATH")
        "Path to store directory",
    Option [] ["state-file"] (ReqArg (\s cfg -> cfg { daemonStateFile = s }) "PATH")
        "Path to daemon state file",
    Option [] ["tmp-dir"] (ReqArg (\t cfg -> cfg { daemonTmpDir = t }) "PATH")
        "Path for temporary files",
    Option ['j'] ["jobs"] (ReqArg (\j cfg -> cfg { daemonMaxJobs = read j }) "N")
        "Maximum concurrent build jobs",
    Option [] ["timeout"] (ReqArg (\t cfg -> cfg { daemonBuildTimeout = read t }) "SECONDS")
        "Build timeout in seconds (0 = no timeout)",
    Option ['k'] ["keep-failed"] (NoArg (\cfg -> cfg { daemonKeepFailed = True }))
        "Keep failed build outputs",
    Option ['u'] ["user"] (ReqArg (\u cfg -> cfg { daemonUser = Just (T.pack u) }) "USER")
        "User to run daemon as",
    Option ['g'] ["group"] (ReqArg (\g cfg -> cfg { daemonGroup = Just (T.pack g) }) "GROUP")
        "Group to run daemon as",
    Option [] ["allowed-users"] (ReqArg (\us cfg -> cfg {
            daemonAllowedUsers = Set.fromList $ map T.strip $ map T.pack $ splitOn ',' us
        }) "USER1,USER2,...")
        "Users allowed to connect to daemon",
    Option [] ["no-auth"] (NoArg (\cfg -> cfg { daemonRequireAuth = False }))
        "Disable authentication requirement",
    Option [] ["allow-privilege-escalation"] (NoArg (\cfg -> cfg { daemonAllowPrivilegeEscalation = True }))
        "Allow users to escalate privileges (dangerous)",
    Option [] ["privilege-model"] (ReqArg (\m cfg -> cfg {
            daemonPrivilegeModel = parsePrivilegeModel m
        }) "always-drop|selective|no-drop")
        "Privilege model (how to handle privilege dropping)",
    Option [] ["log-file"] (ReqArg (\f cfg -> cfg {
            daemonLogFile = if f == "stdout" then Nothing else Just f
        }) "PATH")
        "Path to log file (stdout = log to standard output)",
    Option ['q'] ["quiet"] (NoArg (\cfg -> cfg { daemonLogLevel = LogQuiet }))
        "Minimal logging",
    Option ['v'] ["verbose"] (NoArg (\cfg -> cfg { daemonLogLevel = LogVerbose }))
        "Verbose logging",
    Option ['d'] ["debug"] (NoArg (\cfg -> cfg { daemonLogLevel = LogDebug }))
```

```
        "Debug logging",
    Option ['f'] ["foreground"] (NoArg (\cfg -> cfg { daemonForeground = True }))
        "Run in foreground (don't daemonize)",
    Option [] ["no-auto-start"] (NoArg (\cfg -> cfg { daemonAutoStart = False }))
        "Don't auto-start daemon when client connects",
    Option ['r'] ["restrictive"] (NoArg (\cfg -> cfg { daemonRestrictive = True }))
        "Use more restrictive sandbox settings"
    ]

-- | Parse daemon configuration from command-line arguments
parseConfigFromArgs ::
    SPrivilegeTier t ->            -- ^ Privilege evidence
    [String] ->                    -- ^ Command-line arguments
    TenM 'Build t (Either String DaemonConfig)
parseConfigFromArgs st args = TenM $ \_ _ -> do
    -- Get default configuration
    defaultCfg <- getDefaultConfig

    -- Parse command-line arguments
    case getOpt Permute configOptionDescriptions args of
        (options, _, []) -> do
            -- Apply options to default configuration
            let config = foldl (flip id) defaultCfg options

            -- Validate the resulting configuration
            return $ validateConfig config

        (_, _, errors) ->
            return $ Left $ concat errors

-- | Validate a daemon configuration
validateConfig :: DaemonConfig -> Either String DaemonConfig
validateConfig config = do
    -- Check for required fields
    when (null $ daemonSocketPath config) $
        Left "Socket path cannot be empty"

    when (null $ daemonStorePath config) $
        Left "Store path cannot be empty"

    when (daemonMaxJobs config <= 0) $
        Left "Maximum jobs must be positive"

    when (daemonBuildTimeout config < 0) $
        Left "Build timeout cannot be negative"

    -- Check if privilege escalation is configured safely
    when (daemonAllowPrivilegeEscalation config && not (daemonRequireAuth config)) $
        Left "Cannot allow privilege escalation without authentication"

    -- Check port range if specified
    case daemonPort config of
        Just port | port <= 0 || port > 65535 ->
            Left "Port number must be between 1 and 65535"
        _ -> Right ()

    -- Return the validated config
    return config

-- | Helper to lookup environment variable as Int
lookupEnvInt :: String -> IO (Maybe Int)
lookupEnvInt name = do
    mValue <- lookupEnv name
    case mValue of
        Nothing -> return Nothing
        Just value -> case reads value of
            [(num, "")] -> return $ Just num
```

```haskell
            _ -> return Nothing

-- | Helper to lookup environment variable as Bool
lookupEnvBool :: String -> IO (Maybe Bool)
lookupEnvBool name = do
    mValue <- lookupEnv name
    case mValue of
        Nothing -> return Nothing
        Just value -> case toLower value of
            "true" -> return $ Just True
            "yes" -> return $ Just True
            "1" -> return $ Just True
            "false" -> return $ Just False
            "no" -> return $ Just False
            "0" -> return $ Just False
            _ -> return Nothing

-- | Helper to lookup environment variable as Text
lookupEnvText :: String -> IO (Maybe Text)
lookupEnvText name = do
    mValue <- lookupEnv name
    return $ T.pack <$> mValue

-- | Helper to lookup environment variable as Text list
lookupEnvTextList :: String -> IO (Maybe [Text])
lookupEnvTextList name = do
    mValue <- lookupEnv name
    return $ case mValue of
        Nothing -> Nothing
        Just value -> Just $ map T.strip $ map T.pack $ splitOn ',' value

-- | Helper to lookup environment variable as LogLevel
lookupEnvLogLevel :: String -> IO (Maybe LogLevel)
lookupEnvLogLevel name = do
    mValue <- lookupEnv name
    case mValue of
        Nothing -> return Nothing
        Just value -> case toLower value of
            "quiet" -> return $ Just LogQuiet
            "normal" -> return $ Just LogNormal
            "verbose" -> return $ Just LogVerbose
            "debug" -> return $ Just LogDebug
            _ -> case reads value of
                [(num, "")] | num >= 0 && num <= 3 ->
                    return $ Just $ toEnum num
                _ -> return Nothing

-- | Helper to lookup environment variable as PrivilegeModel
lookupEnvPrivModel :: String -> IO (Maybe PrivilegeModel)
lookupEnvPrivModel name = do
    mValue <- lookupEnv name
    case mValue of
        Nothing -> return Nothing
        Just value -> return $ Just $ parsePrivilegeModel value

-- | Parse privilege model from string
parsePrivilegeModel :: String -> PrivilegeModel
parsePrivilegeModel s = case toLower s of
    "always-drop" -> PrivilegeAlwaysDrop
    "selective" -> PrivilegeDropSelective
    "no-drop" -> PrivilegeNoDrop
    _ -> PrivilegeAlwaysDrop  -- Default to safest option

-- | String utilities
splitOn :: Char -> String -> [String]
splitOn c s = case break (== c) s of
    (chunk, []) -> [chunk]
```

```haskell
    (chunk, _:rest) -> chunk : splitOn c rest

toLower :: String -> String
toLower = map (\c -> if c >= 'A' && c <= 'Z' then toEnum (fromEnum c + 32) else c)

-- | Helper for optional values
(<|>) :: Maybe a -> Maybe a -> Maybe a
Nothing <|> x = x
x <|> _ = x
```

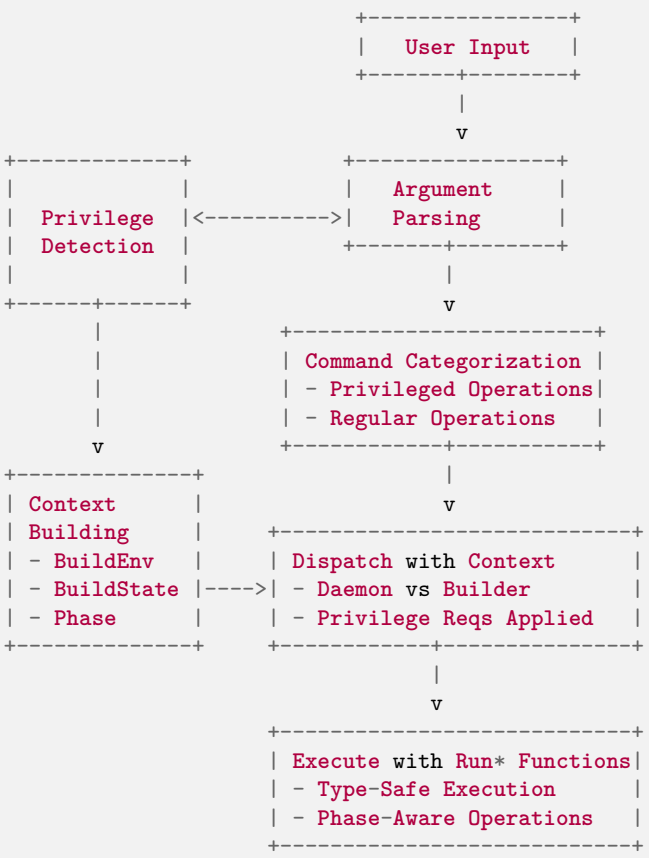
3 CLI Architecture Integration

3.1 Command-Line Interface Architecture

The CLI provides the entry point for users interacting with Ten. It must properly respect and enforce privilege boundaries while providing a clean interface that abstracts the complexity of the underlying type system.

CLI Privilege Enforcement Architecture

```haskell
-- The CLI must respect privilege boundaries at the type level
-- while providing a seamless experience to users
```



4 Runtime Boundary Checks

4.1 Process Communication

Process Communication Boundaries

```haskell
-- In src/Ten/Daemon/Server.hs

-- Validate client request with proper authentication
handleClientRequest ::
    SPrivilegeTier 'Daemon ->
    Request -> TenM p 'Daemon Response
handleClientRequest st request = TenM $ \sp _ -> do
    -- 1. Validate authentication token
    validToken <- validateAuthToken (requestToken request)
    unless validToken $
        throwError $ AuthError "Invalid or expired token"

    -- 2. Check operation permissions
    hasPermission <- checkOperationPermission
                        (requestUserId request)
                        (requestOperation request)
    unless hasPermission $
        throwError $ PrivilegeError "Operation not permitted for this user"

    -- 3. Sanitize all inputs
    sanitizedRequest <- sanitizeRequestInputs request

    -- 4. Process the request
    processRequest sanitizedRequest
```

4.2 External Input Validation

External Input Validation

```haskell
-- In src/Ten/Daemon/Protocol.hs

-- Validate all external inputs before processing
validateExternalInput ::
    SPrivilegeTier 'Daemon ->
    ExternalInput -> TenM p 'Daemon ValidatedInput
validateExternalInput st input = TenM $ \sp _ -> do
    -- 1. Validate file paths to prevent path traversal
    case inputType input of
        FilePath path -> do
            validPath <- isValidFilePath path
            unless validPath $
                throwError $ SecurityError "Invalid file path: potential traversal"

        -- 2. Check for malicious content
        Content content -> do
            isSafe <- scanForMaliciousContent content
            unless isSafe $
                throwError $ SecurityError "Potentially malicious content detected"

        -- 3. Validate URLs/URIs
        URL url -> do
            validURL <- isValidURL url
            unless validURL $
                throwError $ SecurityError "Invalid or disallowed URL"

    -- Return sanitized input
    return $ sanitizeInput input
```

4.3 Derivation Handoff

Derivation Handoff Validation

```haskell
-- In src/Ten/Build.hs

-- Verify derivation when transitioning between build phases
verifyReturnedDerivation ::
    SPrivilegeTier t ->
    Derivation -> FilePath -> TenM 'Build t Derivation
verifyReturnedDerivation st derivation returnPath = TenM $ \sp _ -> do
    -- 1. Verify file exists and is readable
    fileExists <- liftIO $ doesFileExist returnPath
    unless fileExists $
        throwError $ BuildFailed "Return derivation file does not exist"

    -- 2. Check file permissions
    permissions <- liftIO $ getPermissions returnPath
    unless (readable permissions) $
        throwError $ BuildFailed "Return derivation file is not readable"

    -- 3. Read and parse the derivation
    content <- liftIO $ BS.readFile returnPath
    case deserializeDerivation content of
        Left err ->
            throwError $ SerializationError err

        Right innerDrv -> do
            -- 4. Verify derivation hash/signature
            validHash <- verifyDerivationHash innerDrv
            unless validHash $
                throwError $ SecurityError "Return derivation hash verification failed"

            -- 5. Check for cycles in build chain
            chain <- gets buildChain
            let hasCycle = detectCycle (innerDrv : chain)
            when hasCycle $
                throwError $ CyclicDependency "Recursion cycle detected in build chain"

            -- Return the verified derivation
            return innerDrv
```

5 Implementation Strategy

5.1 Phase 1: Core Type System

1. Implement the core singleton types in `Ten/Core.hs`
2. Set up the `TenM` monad with proper phantom type parameters
3. Implement the runtime evidence passing mechanism
4. Create helper functions for type-safe operations

5.2 Phase 2: Store and DB Access

1. Update store operations with proper privilege checks
2. Implement context-aware store access
3. Update database operations with daemon-only constraints
4. Create protocol-based operations for builder context

5.3 Phase 3: Build System

1. Implement proper builder/daemon separation
2. Update sandbox creation with privilege controls
3. Implement derivation handling with phase transitions
4. Create safe return-continuation mechanism

5.4 Phase 4: Integration and Verification

1. Add proper boundary checks at all interfaces
2. Implement protocol mechanism for builder-daemon communication
3. Create helpers for user-friendly syntax
4. Add runtime validation to complement compile-time checks

6 API Design Principles

1. **Universal Syntax**: Maintain clean do-notation for all build patterns
2. **Type Safety**: Provide compile-time guarantees for privilege separation
3. **Runtime Evidence**: Use singletons for dynamic checks
4. **Secure Boundaries**: Ensure proper validation at all trust boundaries
5. **Nix-like Structure**: Follow Nix's architectural patterns with improved type safety

7 Universal Syntax

The core design principle of Ten is providing a single, consistent syntax for all build patterns while handling the complexity internally.

Universal Syntax Pattern

```
-- The universal pattern for ALL build types
build = do
  result <- derivation {
    name = "package",
    builder = "${bash}/bin/bash",
    args = [...],
    ...
  }
  return result
```

7.1 Implementation

Core DSL Implementation

```
-- The universal derivation entry point (public API)
derivation :: forall (t :: PrivilegeTier).
    (SingI t) =>
    DerivationAttrs -> TenM 'Eval t Derivation
derivation attrs = TenM $ \sEval st -> do
    env <- ask

    -- 1. Create the derivation instance
    drv <- createDerivationObject attrs

    -- 2. Analyze dependencies to determine build strategy
    deps <- analyzeDependencies drv

    -- 3. Automatically determine optimal build strategy
    buildStrategy <- if hasReturnContinuationMarkers drv
        then return MonadicStrategy  -- For return-continuation pattern
        else if hasDynamicDependencies drv deps
            then return MonadicStrategy  -- For true monadic dependencies
            else return ApplicativeStrategy  -- For independent builds

    -- 4. Record the strategy in the derivation
    let drv' = drv { derivStrategy = buildStrategy }

    -- 5. Register the derivation in the evaluation graph
    registerDerivation drv'

    return drv'

-- Detect if a derivation contains dynamic dependencies on other derivations
hasDynamicDependencies :: Derivation -> Set Derivation -> TenM 'Eval t Bool
hasDynamicDependencies drv deps = do
    -- Inspect environment variables and arguments for references to other derivations
    -- Detect if builder output is needed by another derivation (sequential dependency)
    -- Return True if dependencies necessitate sequential building
    ...

-- Detect if a derivation uses return-continuation pattern
hasReturnContinuationMarkers :: Derivation -> TenM 'Eval t Bool
hasReturnContinuationMarkers drv = do
    -- Check if builder might produce a continuation derivation:
    --   - Contains bootstrapping or stage indicators
    --   - Uses special return environment variables
    --   - Has builder that's known to produce new derivations
    ...

-- The internal TenM instance implements Monad, Applicative, and Alternative
```

```haskell
-- to automatically optimise build graph based on dependency structure
instance Monad (TenM 'Eval t) where
    (>>=) :: TenM 'Eval t a -> (a -> TenM 'Eval t b) -> TenM 'Eval t b
    (TenM m) >>= f = TenM $ \sp st -> do
        a <- m sp st
        let (TenM m') = f a
        -- Register monadic dependency in graph for proper sequencing
        recordDependency a
        m' sp st

-- Supporting applicative optimisation even in do-notation
instance Applicative (TenM 'Eval t) where
    pure a = TenM $ \_ _ -> pure a

    -- Parallel application when dependencies aren't related
    (<*>) :: TenM 'Eval t (a -> b) -> TenM 'Eval t a -> TenM 'Eval t b
    (TenM mf) <*> (TenM ma) = TenM $ \sp st -> do
        -- Record that these operations can happen in parallel
        recordParallelOperations
        f <- mf sp st
        a <- ma sp st
        return (f a)
```

7.2 Pattern Examples

Three Core Patterns with Identical Syntax

```
-- 1. APPLICATIVE (Parallel) Pattern
buildParallel = do
  lib <- derivation { name = "lib", ... }
  docs <- derivation { name = "docs", ... }
  tests <- derivation { name = "tests", ... }

  -- Ten detects these are independent and builds in parallel
  return { library = lib, documentation = docs, tests = tests }

-- 2. MONADIC (Sequential) Pattern
buildSequential = do
  lib <- derivation { name = "library", ... }

  -- This explicitly depends on lib, forcing sequential execution
  app <- derivation {
    name = "application",
    builder = "${gcc}/bin/gcc",
    args = ["-L${lib}/lib", "-o", "$out/bin/app", "main.c"]
  }

  return app

-- 3. RETURN-CONTINUATION Pattern
buildBootstrap = do
  stage1 <- derivation {
    name = "compiler-bootstrap",
    builder = "${gcc}/bin/gcc",
    args = ["-o", "$out/bin/compile", "compiler.c"]
  }

  -- Ten automatically checks if stage1's builder wrote a
  -- new derivation to $TEN_RETURN_PATH and builds it next
  return stage1
```

7.3 Benefits Over Nix

Advantages Over Nix's Multiple Patterns

```
-- In Nix, users must manually choose between different patterns:

-- Nix APPLICATIVE pattern (callPackage)
app = callPackage ./app.nix { inherit lib; }

-- Nix MONADIC pattern (import)
app = import ./app.nix { inherit lib; }

-- Nix RETURN-CONTINUATION pattern (override)
compiler = (import ./bootstrap.nix {}).overrideAttrs (old: { ... })

-- In Ten, ALL patterns use the same do-notation syntax:
anyBuild = do
  result <- derivation { ... }
  return result

-- The system automatically determines the optimal build strategy:
-- 1. Analyzes data dependencies between derivations
-- 2. Selects parallel building when possible
-- 3. Detects return-continuation pattern automatically
-- 4. Optimises using content-addressed storage
```