```
/*
*       Crypto for Bitwarden against openSSL.
*
*/
extern crate hmac;
extern crate openssl;
extern crate rustc_serialize as serialize;
use serialize::base64;
// use serialize::base64::FromBase64;
 use serialize::base64::ToBase64;
// use std::io;

pub fn hmac_openssl(key: &[u8], messages: &str) -> String {
    let pkey = openssl::pkey::PKey::hmac(&key).unwrap();
    let mut signer = openssl::sign::Signer::new(openssl::hash::MessageDigest::sha256(), &pkey).unwrap();
    signer.update(messages.as_bytes()).unwrap();
    let mut result = String::from("");
    result.push_str(&signer.sign_to_vec().unwrap().to_base64(base64::STANDARD)[..]);
    return result;
}

//make a master key.
pub fn make_key(password: &str, salt: &str) -> [u8; 32] {
    // 256-bit derived key
    //  hashlib.pbkdf2_hmac(’sha256’, password, salt, 5000, dklen=32)
    //let mut dk = [0u8; 32];
    //let mut dk = [0u8; 32];
    let mut derived_key= [0; 32];
    openssl::pkcs5::pbkdf2_hmac(&password.as_bytes(),salt.as_bytes(), 5000, openssl::hash::MessageDigest::sha256(), &mut derived_key).unwrap();
    //let mut result = String::from("");
    //result.push_str(&derived_key.to_base64(base64::STANDARD)[..]);
    return derived_key;
}

//# base64-encode a wrapped, stretched password+salt for signup/login
pub fn hashed_password(password: &str, salt: &str) -> String {
    let key = make_key(password, salt);
    //let mut derived_key = [0u8; 32];
    let mut derived_key = [0; 32];
    openssl::pkcs5::pbkdf2_hmac(&key,&password.as_bytes(), 1, openssl::hash::MessageDigest::sha256(), &mut derived_key).unwrap();
    let mut result = String::from("");
    result.push_str(&derived_key.to_base64(base64::STANDARD)[..]);
    return result;
}

// encode into a bitwarden compatible cipher string.
pub fn encode_cipher_string(enctype: &u8, iv: &[u8], ct: &[u8], mac: &[u8]) -> String {
    let mut result = String::from("");
    result.push_str(&enctype.to_string());
    result.push_str(".");
    if iv.len() > 0 {
        result.push_str(&iv.to_base64(base64::STANDARD)[..]);
    }
    if ct.len() > 0 {
        result.push_str("|");
        result.push_str(&ct.to_base64(base64::STANDARD)[..]);
    }
    if mac.len() > 0 {
        result.push_str("|");
        result.push_str(&mac.to_base64(base64::STANDARD)[..]);
    }
    return result;
}
pub struct Cipherstring {
    encryption_type: u8,
    iv: Vec<u8>,
    ct: Vec<u8>,
    mac: Vec<u8>,
}

//decode a bitwarden cipher string
pub fn decode_cipher_string(cipher_string: &str) -> Cipherstring {
    let pieces: Vec<&str> = cipher_string.split("|").collect();
    let beg = pieces[0];
    let beg_pieces: Vec<&str> = beg.split(".").collect();
    let enc_type = beg_pieces[0];
    let iv = beg_pieces[1];
    let ct = pieces[1];
    let mut mac = vec![0; 0];
    if pieces.len() == 3 {
        mac = pieces[2].from_base64().unwrap();
    } else {
        mac = [0; 0].to_vec();
    };
    let result = Cipherstring {
        encryption_type: enc_type.parse::<u8>().unwrap(),
        iv: iv.from_base64().unwrap(),
        ct: ct.from_base64().unwrap(),
        mac,
    };
    return result;
}

//create symmetric key (encryption_key and mac_key from secure random bytes
// we return it as one large 64 byte variable, the first 32 are the encryption key and the last 32 are the mac_key.
pub fn symmetric_key() -> (Vec<u8>) {
    let mut rng = rand::thread_rng();
    let encryption_key: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 32).unwrap();
    let mac_key: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 32).unwrap();
    return encryption_key + mac_key;
}

// make encryption key
pub fn make_encrypted_key(symmetric_key: Vec<u8>, master_key: [u8; 32]) -> String {
    let mut rng = rand::thread_rng();
    let iv: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 16).unwrap();
    let cipher = encrypt_aes_256_cbc(&symmetric_key, &master_key, &iv).unwrap();
    let mac: [u8; 0] = [];
    let ret = encode_cipher_string(&0, &iv, &cipher, &mac);
    return ret;
}

//double hmac compare.
pub fn macs_equal(mac_key: &[u8], mac1: &[u8], mac2: &[u8]) -> bool {
    let mut hmac1 = hmac::Hmac::new(crypto::sha2::Sha256::new(), &mac_key);
    hmac1.input(&mac1);
    let mut hmac2 = hmac::Hmac::new(crypto::sha2::Sha256::new(), &mac_key);
    hmac2.input(&mac2);
    return hmac1.result() == hmac2.result();
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_hashed_password() {
        let result = hashed_password("password", "nobody@example.com");
        let expected = "2cj6A0brDusMjVlVqcBW2a+kiOQDqZDCEB40NshJE7o=";
        assert_eq!(expected, result);
    }
    #[test]
    fn test_make_key() {
        let expected = b"\x95\xa9\xc3\xb6W\xfb\xa7r\x80\xbfY\xdf\xfc\x18S\x81\x9e+\xf7W\xd0\x1db\x92$\x1bN\x05\xf5\xb8s\xe7";
        let result = make_key("password", "nobody@example.com");
        assert_eq!(expected, &result);
    }
    #[test]
    fn test_decrypt_encrypted_key() {
        let expected = b"";
        //let result = decrypt_encrypted_key("0.QjjRqI96zTTB7/z3wHInzg==|WHl3wQjcPmZJ4wgADXywOhMB6RILrqPcivCJc50OkivznCRaFTBXVe6MudDxYcJEu6M7RMVQfz71LEcmcy/DFOT5veHR9YCdp4kQj3t4Tx0=",);
        //assert_eq!(expected, &result);
    }
}
```