

Feb 27, 19 8:51

lib.rs

Page 1/5

```

1  extern crate crypto;
2  extern crate rand;
3  extern crate reqwest;
4  extern crate rustc_serialize as serialize;
5  extern crate uuid;
6
7  #[macro_use]
8  extern crate serde_json;
9
10 use crypto::buffer::{ReadBuffer, WriteBuffer};
11 use crypto::hmac;
12 use crypto::mac::Mac;
13 use crypto::symmetriccipher;
14 use serialize::base64;
15 use serialize::base64::FromBase64;
16 use serialize::base64::ToBase64;
17 use std::io;
18 //use reqwest::header::{Authorization, Basic};
19 use reqwest::header;
20
21 //make a master key.
22 pub fn make_key(password: &str, salt: &str) -> io::Result<[u8; 32]> {
23     // 256-bit derived key
24     // hashlib.pbkdf2_hmac('sha256', password, salt, 5000, dklen=32)
25     let mut dk = [0u8; 32];
26     let mut mac =
27         crypto::hmac::Hmac::new(crypto::sha2::Sha256::new(), &password.as_bytes().to_vec());
28     let count = 5000;
29     crypto::pbkdf2::pbkdf2(&mut mac, &salt.as_bytes().to_vec(), count, &mut dk);
30     return Ok(dk);
31 }
32
33 ///# base64-encode a wrapped, stretched password+salt for signup/login
34 pub fn hashed_password(password: &str, salt: &str) -> String {
35     let key = make_key(password, salt);
36     let mut derived_key = [0u8; 32];
37     let mut mac = crypto::hmac::Hmac::new(crypto::sha2::Sha256::new(), &key.unwrap().to_vec());
38     let count = 1;
39     crypto::pbkdf2::pbkdf2(
40         &mut mac,
41         &password.as_bytes().to_vec(),
42         count,
43         &mut derived_key,
44     );
45     let mut result = String::from("");
46     result.push_str(&derived_key.to_base64(base64::STANDARD)[..]);
47     return result;
48 }
49
50 // encode into a bitwarden compatible cipher string.
51 pub fn encode_cipher_string(etype: &u8, iv: &[u8], ct: &[u8], mac: &[u8]) -> String {
52     let mut result = String::from("");
53     result.push_str(&etype.to_string());
54     result.push_str(".");
55     if iv.len() > 0 {
56         result.push_str(&iv.to_base64(base64::STANDARD)[..]);
57     }
58     if ct.len() > 0 {
59         result.push_str("|");
60         result.push_str(&ct.to_base64(base64::STANDARD)[..]);
61     }
62     if mac.len() > 0 {
63         result.push_str("|");
64         result.push_str(&mac.to_base64(base64::STANDARD)[..]);
65     }
66     return result;
67 }
68 pub struct Cipherstring {
69     encryption_type: u8,
70     iv: Vec<u8>,
71     ct: Vec<u8>,
72     mac: Vec<u8>,
73 }
74
75 //decode a bitwarden cipher string
76 pub fn decode_cipher_string(cipher_string: &str) -> Cipherstring {
77     let pieces: Vec<&str> = cipher_string.split("|").collect();
78     let beg = pieces[0];
79     let beg_pieces: Vec<&str> = beg.split(".").collect();
80     let enc_type = beg_pieces[0];
81     let iv = beg_pieces[1];
82     let ct = pieces[1];
83     let mut mac = vec![0; 0];
84     if pieces.len() == 3 {
85         mac = pieces[2].from_base64().unwrap();
86     } else {

```

Feb 27, 19 8:51

lib.rs

Page 2/5

```

87     mac = [0; 0].to_vec();
88 };
89 let result = Cipherstring {
90     encryption_type: enc_type.parse::<u8>().unwrap(),
91     iv: iv.from_base64().unwrap(),
92     ct: ct.from_base64().unwrap(),
93     mac,
94 };
95 return result;
96 }
97
98 //create symmetric key (encryption_key and mac_key from secure random bytes
99 pub fn symmetric_key() -> (Vec<u8>, Vec<u8>) {
100     let mut rng = rand::thread_rng();
101     let encryption_key: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 32).unwrap();
102     let mac_key: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 32).unwrap();
103     return (encryption_key, mac_key);
104 }
105
106 // make encryption key
107 pub fn make_encrypted_key(symmetric_key: Vec<u8>, master_key: [u8; 32]) -> String {
108     let mut rng = rand::thread_rng();
109     let iv: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 16).unwrap();
110     let cipher = encrypt_aes_256_cbc(&symmetric_key, &master_key, &iv).unwrap();
111     let mac: [u8; 0] = [];
112     let ret = encode_cipher_string(&0, &iv, &cipher, &mac);
113     return ret;
114 }
115
116 //double hmac compare.
117 pub fn macs_equal(mac_key: &[u8], mac1: &[u8], mac2: &[u8]) -> bool {
118     let mut hmac1 = hmac::Hmac::new(crypto::sha2::Sha256::new(), &mac_key);
119     hmac1.input(&mac1);
120     let mut hmac2 = hmac::Hmac::new(crypto::sha2::Sha256::new(), &mac_key);
121     hmac2.input(&mac2);
122     return hmac1.result() == hmac2.result();
123 }
124
125 //decrypt encryption key
126 //pub fn decrypt_encrypted_key(cipher_string: &str, key: &[u8], mac_key: &[u8]) -> ( Vec<u8>, Vec<u8> ) {
127 //     let cipher_struct = decode_cipher_string(cipher_string);
128 //     let iv = cipher_struct.iv.drain(..).collect();
129 //     let encrypted_data = cipher_struct.ct.drain(..).collect();
130 //     assert_eq!(cipher_struct.encryption_type, 0);
131 //     let mut decryptor = crypto::aes::cbc_decryptor(
132 //         crypto::aes::KeySize::KeySize256,
133 //         key,
134 //         iv,
135 //         crypto::blockmodes::PkcsPadding,
136 //     );
137 //     let mut final_result = Vec::<u8>::new();
138 //     let mut read_buffer = crypto::buffer::RefReadBuffer::new(encrypted_data);
139 //     let mut buffer = [0; 4096];
140 //     let mut write_buffer = crypto::buffer::RefWriteBuffer::new(&mut buffer);
141 //
142 //     loop {
143 //         let result = try!(decryptor.decrypt(&mut read_buffer, &mut write_buffer, true));
144 //         final_result.extend(
145 //             write_buffer
146 //                 .take_read_buffer()
147 //                 .take_remaining()
148 //                 .iter()
149 //                 .map(|&i| i),
150 //         );
151 //         match result {
152 //             crypto::buffer::BufferResult::BufferUnderflow => break,
153 //             crypto::buffer::BufferResult::BufferOverflow => {}
154 //         }
155 //     }
156 //
157 //     let symmetric_key = final_result.drain(0..32).collect();
158 //     let mac_key = final_result.drain(32..0).collect();
159 //     return (symmetric_key, mac_key);
160 // }
161
162 // decrypt AES-256-CBC
163 pub fn decrypt_aes_256_cbc(
164     encrypted_data: &[u8],
165     key: &[u8],
166     iv: &[u8],
167 ) -> Result<Vec<u8>, symmetriccipher::SymmetricCipherError> {
168     let mut decryptor = crypto::aes::cbc_decryptor(
169         crypto::aes::KeySize::KeySize256,
170         key,
171         iv,
172         crypto::blockmodes::PkcsPadding,

```

Feb 27, 19 8:51

lib.rs

Page 3/5

```

173     );
174
175     let mut final_result = Vec::<u8>::new();
176     let mut read_buffer = crypto::buffer::RefReadBuffer::new(encrypted_data);
177     let mut buffer = [0; 4096];
178     let mut write_buffer = crypto::buffer::RefWriteBuffer::new(&mut buffer);
179
180     loop {
181         let result = try!(decryptor.decrypt(&mut read_buffer, &mut write_buffer, true));
182         final_result.extend(
183             write_buffer
184                 .take_read_buffer()
185                 .take_remaining()
186                 .iter()
187                 .map(|&i| i),
188         );
189         match result {
190             crypto::buffer::BufferResult::BufferUnderflow => break,
191             crypto::buffer::BufferResult::BufferOverflow => {}
192         }
193     }
194
195     Ok(final_result)
196 }
197 // encrypt AES-256-CBC
198 pub fn encrypt_aes_256_cbc(
199     data: &[u8],
200     key: &[u8],
201     iv: &[u8],
202 ) -> Result<Vec<u8>, symmetriccipher::SymmetricCipherError> {
203     //setup
204     let mut final_result = Vec::<u8>::new();
205     let mut read_buffer = crypto::buffer::RefReadBuffer::new(data);
206     let mut buffer = [0; 4096];
207     let mut write_buffer = crypto::buffer::RefWriteBuffer::new(&mut buffer);
208     let mut encryptor = crypto::aes::cbc_encryptor(
209         crypto::aes::KeySize::KeySize256,
210         key,
211         iv,
212         crypto::blockmodes::PkcsPadding,
213     );
214     loop {
215         let result = try!(encryptor.encrypt(&mut read_buffer, &mut write_buffer, true));
216
217         // "write_buffer.take_read_buffer().take_remaining()" means:
218         // from the writable buffer, create a new readable buffer which
219         // contains all data that has been written, and then access all
220         // of that data as a slice.
221         final_result.extend(
222             write_buffer
223                 .take_read_buffer()
224                 .take_remaining()
225                 .iter()
226                 .map(|&i| i),
227         );
228
229         match result {
230             crypto::buffer::BufferResult::BufferUnderflow => break,
231             crypto::buffer::BufferResult::BufferOverflow => {}
232         }
233     }
234
235     Ok(final_result)
236 }
237
238 // api/accounts/register implementation.
239 pub fn register(url: &request::Url, email: &String, password: &String) -> String {
240     let master_password_hash = hashed_password(password, email);
241     let master_key = make_key(password, &email);
242     let (val0, val1) = symmetric_key();
243     let mut sym_key = Vec::new();
244     sym_key.clone_from_slice(&val0);
245     sym_key.clone_from_slice(&val1);
246     let protected_key = make_encrypted_key(sym_key, master_key.unwrap());
247     let signup_json = json!({
248         "name": null,
249         "email": email,
250         "masterPasswordHash": master_password_hash,
251         "masterPasswordHint": null,
252         "key": protected_key,
253     });
254     //let url = request::url::Url::parse(url);
255     //let register_url = format!("{}",api/accounts/register", url);
256     let client = request::Client::new();
257     let mut response = client
258         .post(url.join("/api/accounts/register").unwrap())

```

Feb 27, 19 8:51

lib.rs

Page 4/5

```

259     .body(signup_json.to_string())
260     .send()
261     .unwrap();
262     // println!("register post:{:?}", res);
263     if response.status().is_success() {
264         println!("success!");
265     } else if response.status().is_server_error() {
266         println!("server error!");
267     } else {
268         println!("Something else happened. Status: {:?}", response.status());
269     }
270     let response_body = response.text();
271     return response_body.unwrap();
272 }
273
274 //api//sync
275 pub fn sync(url: &request::Url, access_token: &str) -> String {
276     let mut header = request::header::Headers::new();
277     header.set(request::header::Authorization(request::header::Bearer {
278         token: access_token.to_owned(),
279     }));
280     let client = request::Client::new();
281     let mut response = client
282         .get(url.join("/api/sync").unwrap())
283         .headers(header)
284         .send()
285         .unwrap();
286     return response.text().unwrap();
287 }
288 // api/connect/token implementation.
289 pub fn login(url: &request::Url, email: &String, password: &String) -> String {
290     //let internal_key = make_key(password, &email);
291     let master_password_hash = hashed_password(&password, &email);
292     drop(password);
293     //let id = uuid::Uuid::new_v5(&uuid::NAMESPACE_DNS, "foo").to_string();
294     let id = "49a521be-c920-4cba-b6ba-f170c3993669";
295     //println!("uuid:{}, id);
296     let params = [
297         ("grant_type", "password"),
298         ("username", email),
299         ("password", &master_password_hash),
300         ("scope", "api offline_access"),
301         ("client_id", "browser"),
302         ("deviceType", "3"),
303         ("deviceIdentifier", id),
304         ("deviceName", "firefox"),
305         ("devicePushToken", ""),
306     ];
307     //let url = request::url::Url::parse(url);
308     //let register_url = format!("{}", api/accounts/register", url);
309     let client = request::Client::new();
310     let mut response = client
311         .post(url.join("/identity/connect/token").unwrap())
312         .form(&params)
313         .send()
314         .unwrap();
315     // println!("register post:{:?}", res);
316     if response.status().is_success() {
317         println!("success!");
318     } else if response.status().is_server_error() {
319         println!("server error!");
320     } else {
321         println!("Something else happened. Status: {:?}", response.status());
322     }
323     let response_body = response.text();
324     return response_body.unwrap();
325 }
326
327 #[cfg(test)]
328 mod tests {
329     use super::*;
330     #[test]
331     fn test_hashed_password() {
332         let result = hashed_password("password", "nobody@example.com");
333         let expected = "2cj6A0brDusMjVlVqcBW2a+kiOQDqZDCEB40NshJE7o=";
334         assert_eq!(expected, result);
335     }
336     #[test]
337     fn test_make_key() {
338         let expected = b"\x95\xa9\xc3\xb6W\xfb\xa7r\x80\xbfY\xdf\xfc\x18S\x81\x9e+\xf7W\xd0\x1db\x92$\x1bN\x05\x
339         f5\xb8s\xe7";
340         let result = make_key("password", "nobody@example.com").unwrap();
341         assert_eq!(expected, &result);
342     }
343     #[test]
344     fn test_decrypt_encrypted_key() {

```

Feb 27, 19 8:51

lib.rs

Page 5/5

```
344     let expected = b"";
345     //let result = decrypt_encrypted_key("0.QjjRqI96zTTB7/z3wHInzg=|WHl3wQjcPmZJ4wgADXyWOhMB6RILrqPcivCJc5
00kivznCRaFTBXVe6MudDxYcJEu6M7RMVQfz71LEcmcy/DFOT5veHR9YCdp4kQj3t4Tx0=",);
346     //assert_eq!(expected, &result);
347 }
348 }
349
```