Project17：比较 Firefox 和谷歌的记住密码插件的实现区别

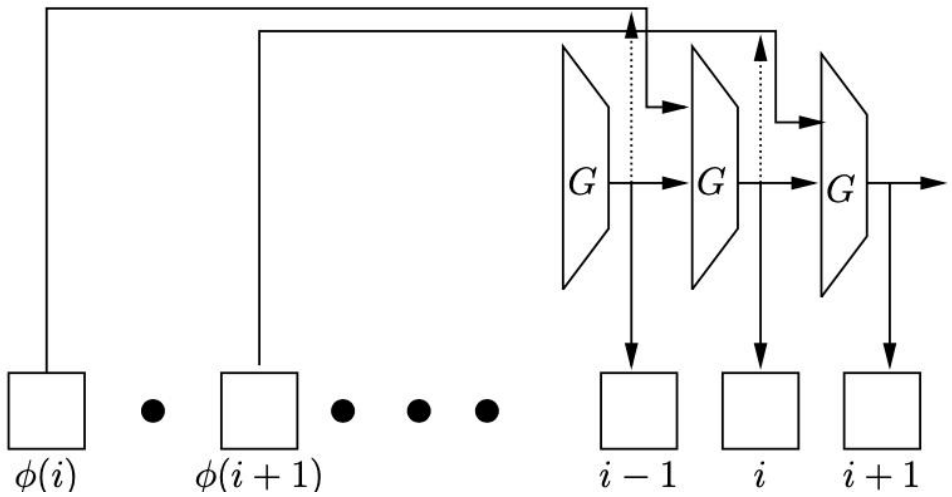
一、密码的用途和储存方法

Web 服务主要通过使用密码进行用户认证，这些密码往往以哈希形式存在于数据库内。然而数据库如果被盗取，经历字典攻击后的数据库可以被轻易破解，原因在于这些密码的熵值较底，有许多用户会在不同的系统中使用相同的密码。

解决方案是在密码哈希过程中加入盐（Salt）。尽管加盐可以解决许多问题，但无法挡住暴力攻击，特别是当哈希计算可以借助涉及 GPU、FPGA、ASIC 等定制硬件进行低成本计算的情况下。密码和盐同时被盗取时，破解的成本将会更低。

解决以上问题的关键在于，当哈希方法需要使用大块内存进行计算时，GPU 和 ASIC 等硬件就无法发挥作用。因此，开发了需要大量内存计算的哈希函数（Memory-hard Hash Function）。
火狐浏览器使用 PBKDF2 作为其哈希函数，而谷歌浏览器使用 Argon2。

二、Argon2d 的使用



谷歌浏览器中所使用的 Argon2d 进行数据相关的内存访问，使其在加密数字货币以及工作量证明的应用上表现优良，同时抵抗侧信道定时攻击。Argon2i 使用数据无关的内存访问，是密码哈希的首选方法。Argon2id 则结合了 Argon2i 和 Argon2d 的优点，旨在提供侧信道攻击保护，同时节省暴力开销。

三、Argon2 的安全性分析

对 Argon2 的分析表明该协议展示出两个主要的安全属性：

“被动”攻击者即使获取了服务器存储的数据库内容，只能做两件事：1. 学习到密码的哈希值 kA。2. 对密码进行依赖硬性哈希计算的暴力攻击。
“主动”攻击者，即那些窃取 TLS 连接信息或者干扰正在运行的密钥服务器的攻击者，可以做的事包括：1. 学习到密码的哈希值 kA。2. 控制账户，也就是说可以生产校验。3. 对密码和哈希值 kB 执行简单暴力攻击，即对每个猜测的密码都需要进行 1000 轮 PBKDF。

尽管 Argon2 相比于基于 SRP 的协议稍显不足，但实际上它的安全性强于大部分业界实践，并且更适合客户端实现。一段长期服务器数据可以阻止简单的字典攻击，即使攻击者获取了所有服务器数据库内容，也需要对每个猜测进行完整的 scrypt 延伸计算。

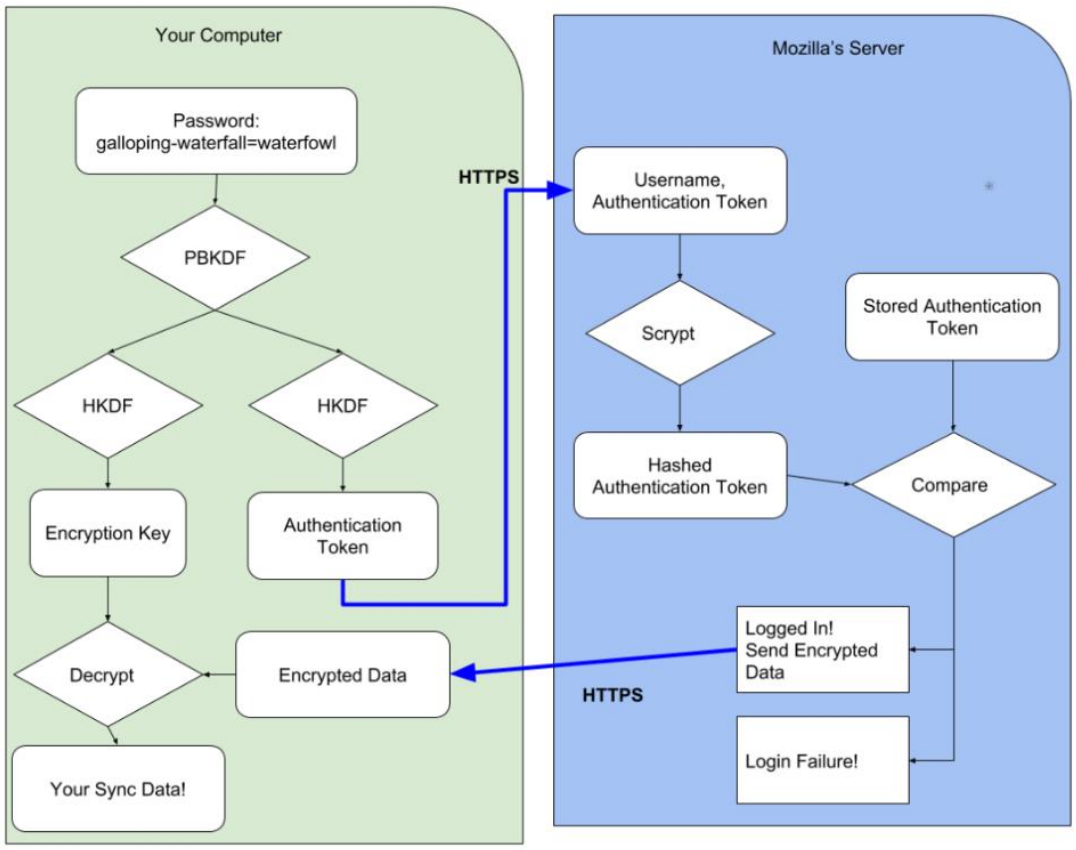
被动攻击者可以通过两个值预测密码猜测。其中一种是“verifyHash”，源自完整的基于 scrypt 的扩展输出。另一种则是对一些 B 类加密数据进行测试密码，这同样受到 scrypt 过程的保护：对于每个密码，攻击者都需要执行完整的计算得出 kB，然后尝试解密一部分数据，并检查其 HMAC 是否一致。

HMAC 检查通过表示候选密码正确。攻击者可以利用预加密响应作为预测机，但服务器明确不会保留加密它的响应。所以，keyFetchToken 是随机生成的，独立于用户的密码，因此加密的数据并不能帮助测试密码猜测。

四、PBKDF2 的使用

PBKDF2 实际上就是将伪散列函数 PRF（pseudorandom function）应用到输入的密码、salt 中，生成一个散列值，然后将这个散列值作为一个加密 key，应用到后续的加密过程中，以此类推，将这个过程重复很多次，从而增加了密码破解的难度，这个过程也被称为是密码加强。

我们看一个标准的 PBKDF2 工作的流程图：



从图中可以看到，初始的密码跟 salt 经过 PRF 的操作生成了一个 key，然后这个 key 作为下一次加密的输入和密码再次经过 PRF 操作，生成了后续的 key，这样重复很多次，生成的 key 再做异或操作，生成了最终的 T，然后把这些最终生成的 T 合并，生成最终的密码。

根据 2000 年的建议，一般来说这个遍历次数要达到 1000 次以上，才算是安全的。当然这个次数也会随着 CPU 计算能力的加强发生变化。这个次数可以根据安全性的要求自行调整。

有了遍历之后，为什么还需要加上 salt 呢？加上 salt 是为了防止对密码进行彩虹表攻击。也就是说攻击者不能预选计算好特定密码的 hash 值，因为不能提前预测，所以安全性得以提高。标准 salt 的长度推荐是 64bits，美国国家标准与技术研究所推荐的 salt 长度是 128 bits。

五、参考文献

> Password Hashing: Scrypt, Bcrypt and ARGON2
> https://github.com/mozilla/fxa-auth-server/wiki/onepw-protocol#vs-old-sync

> PBKDF2 加密_pbkdf2 是对称加密_柏修的博客-CSDN 博客

六、附录:

```
1. public static final int HASH_MILLIS = 1231;
2. public static final String ALGORITHM = "asfdasdfdfsafs";
3. public static final int ITERATION_COUNT = 123123;
4. public static final int KEY_SIZE = 123;
5. public static final int SALT_LENGTH = 123;
6.
7. public static String encryptPassword(String salt,String password) throws Exception{
8.     byte[] saltByte = Base64.decodeBase64(salt.getBytes());
9.     byte[] hash = PasswordsUtils.hashPassword(password.toCharArray(), saltByte);
10.    String pwd_hash_str = new String(Base64.encodeBase64(hash));
11.    return pwd_hash_str;
12. }
13.
14. public static byte[] hashPassword(char[] password, byte[] salt)
15.         throws GeneralSecurityException {
16.     return hashPassword(password, salt, ITERATION_COUNT, KEY_SIZE);
17. }
18.
19. public static byte[] hashPassword(char[] password, byte[] salt,
20.         int iterationCount, int keySize) throws GeneralSecurityException {
21.     try {
22.         PBEKeySpec spec = new PBEKeySpec(password, salt, iterationCount, keySize);
23.         SecretKeyFactory factory = SecretKeyFactory.getInstance(ALGORITHM);
24.         return factory.generateSecret(spec).getEncoded();
25.     } catch (IllegalArgumentException e) {
26.         throw new GeneralSecurityException("key size " + keySize, e);
27.     }
28. }
29.
30. public static boolean matches(char[] password, byte[] passwordHash, byte[] salt)
31.         throws GeneralSecurityException {
32.     return matches(password, passwordHash, salt, ITERATION_COUNT, KEY_SIZE);
33. }
34.
35. public static boolean matches(char[] password, byte[] passwordHash, byte[] salt,
36.         int iterationCount, int keySize) throws GeneralSecurityException {
37.     return Arrays.equals(passwordHash, hashPassword(password, salt,
38.             iterationCount, keySize));
39. }
40.
```

```
41.
42. public static byte[] nextSalt() {
43.     byte[] salt = new byte[SALT_LENGTH];
44.     SecureRandom sr = new SecureRandom();
45.     sr.nextBytes(salt);
46.     return salt;
47. }
48.
```