如何实现用户认证授权系统

December 4, 2013

需求

1. 用户注册
2. 登陆(Authentication)
3. 登出
4. 访问权限控制（Authorization，匿名用户、普通用户、管理员）

实现

Authentication

一旦用户注册之后，用户信息就保存在服务器端（DB/Cache）。关键在于用户需要提供身份凭证，一般是用户名和密码。即常见的登陆页面：用户输入username和password，勾选Remember Me（可选，一般是记住一周），点击登陆，提交请求到服务端（这里一般是走HTTPS）。服务端根据用户名和密码到数据库查询是否匹配，如果匹配的话，说明身份认证成功。这是一次普通的身份认证过程。非常好理解。

关键在于HTTP是无状态的，用户登陆过一次，但是如果你没有做一些状态管理操作的话，用户登陆后请求同一个页面，服务器仍然要求其登陆。这时候就需要做一些状态处理了。一般是通过服务器颁发一个登陆凭证(sessionkey/token/ticket)实现的。那么这个登陆凭证是怎么生成的？又是怎样安全的颁发给客户端呢？

方案一：session集群 + 随机sessionId

用户登陆成功之后，服务器为其随机生成的一个sesionId，保存在session服务器中，将这个sessionId与用户关联起来：sessionId=>userId。然后通过cookies方式将sessionId颁发给客户端，浏览器下次请求会自动带上这个sessionId。服务器根据sessionId从session服务器中拿到关联的userId，比较是否与请求的userId相同，如果是则认为是合法请求。

sessionId是随机生成的，基本来说是不可能被猜测出来的，所以这方面的安全还是有一定保障的。

方案二：session-less + 加密算法

上面的Authentication方式其实是用到了session和cookies。我们知道session这东西是服务端状态，而服务端一旦有状态，就不是很好线性扩展。其实对于身份验证来说，服务端保留的也这是一个简单的value而已，一般是userId，即session['sessionId']==>userId。然后再根据userId去DB获取用户的详细信息。如果我们把userId作为一个cookies值放在客户端，然后把用几个cookies值（比如userId）做一个特殊的签名算法得到的token也放在cookie中，即`f(userId, expireTime)==>token`。这样服务端得到用户请求，用同样的签名算法进行计算，如果得到的token是相等，那么证明这个用户是合法的用户。注意这个签名算法的输入因子必须包含过期时间这样的动态因子，否者得到的token永远是固定的。这种实现方式其实是仿造CSRF防御机制anti-csrf.md。是笔者自己想出来的，不清楚有没有人用过，个人感觉行得通。

然而上面的做法安全性取决于签名算法的隐蔽性，我们可以更进一步的，可以参考API中的签名验证方式，把password作为secretKey，把userId, expireTime, nonce, timestamp作为输入参数，同样用不公开的算法（这个与API签名不同）结合password这个secretKey进行计算，得到一个签名，即`f(userId, expireTime, nonce, timestamp, password)==>sign`。每次客户端传递userId, expireTime, nonce, timestamp和sign值，我们根据userId获取到password，然后进行`f(userId, expireTime, nonce, timestamp, password)==>sign`计算，然后比较两个sign是否一致，如果是，表示通过。这种方式比起上面的方式其实区别在于增加了password作为输入参数。这样首先增加签名的破解难度。还带来一个额外的好处，就是当用户修改了password之后，这个token就失效了，更合理安全一些。

具体步骤如下：

1. 客户端 >>>

1. 用户输入userId和password，form表单提交到后台进行登录验证（这里最好走HTTPS）。

2. <<< 服务端

1. 服务端根据userId，从用户信息表中得到用户注册时保存的密码签名：S=md5(password)。
2. 服务器验证用户信息：userId == DB.userId; md5(password) == DB.S。
3. 如果验证通过，说明用户身份认证通过。这时候服务器会为客户端分配一个票据(签名)：`token=md5(userId;tokenExpiryTime;S;secretKey)`。其中secretKey是一个后台统一的密钥，而且跟DB是分开的，一般是写在配置文件中。目的很明显，就是避免将鸡蛋放在同一个篮子中。然后服务端将这个token值存放在cookies中。同样放入cookies中的还有userId和tokenExpiryTime。这些cookies的过期时间都是tokenExpiryTime。

3. 客户端 >>>

1. 客户端每次请求都要带上这个三个cookies(浏览器自动会带上)。

4. <<< 服务端

1. 服务器首先检查tokenExpiryTime是否过期了，如果过期就要求用户重新登录。否则，根据userId cookie从用户信息表中获取用户信息。然后计算 `expectedToken=md5(userId;tokenExpiryTime;S;secretKey)`。然后比较expectedToken是否跟用户提交的token相同，如果相同，表示验证通过；否则表示验证失败。

说明

1. 为了增加安全性，可以进一步将userId, tokenExpiryTime; token 这个三个cookies进行一次对称加密: ticket=E(userId:tokenExpiryTime:token)。
2. 虽然cookies的值是没有加密的，但是由于有签名的校验。如果黑客修改了cookie的内容，但是由于他没有签名密钥secretKey，会导致签名不一致。

google了一下，发现这篇文章跟我的观点不谋而合Sessionless_Authentication_with_Encrypted_Tokens。另外看了一下Spring Security的Remember Me实现，原来这种方式是5.2. Simple Hash-Based Token Approach方式。他的hash因子中也有password，这样当用户修改了password之后，这个token就失效了。

这种基于password和secretKey做token的鉴权方式实现非常简单，而且客户端没有任何计算和验证逻辑，非常适合于BS架构的。不过这种方式在安全性方面还有一些问题：

1. 客户端没法验证服务器的真实性。域名劫持的情况很容易伪造服务器。
2. token放在cookies中，还是容易被盗取（比如XSS漏洞，或者网络窃听）。使用动态token可以避免这个问题，但是需要持久化token，比较麻烦，而且对性能有消耗5.3 Persistent Token Approach。

一个简单而有效的解决方案就是使用HTTPS。HTTPS使用CA证书验证服务器的合法性，全程会话（包括cookies）都是经过加密传输，刚好解决了上面的两个安全问题。很多网站都是使用这种鉴权认证方案。比如GitHub。不过安全性要求不是很高的网站还是采用的是登陆认证的时候HTTPS，其他情况HTTP的方式，比如新浪微博、亚马逊、淘宝、quora等。

TIPS SSO

上面的鉴权方式依赖于Cookies来传递sessionId，而我们知道Cookies具有跨域限制。不过可以通过一些方式解决。具体可以看一下这篇文章，写的非常好。Building and implementing a Single Sign-On solution。

上面的想法是把password作为一个secretKey或者Salt进行签名。还有另一种思路，就是把password作为对称密钥来进行加密解密。具体步骤如下：

1. 客户端 >>>

1. 用户输入userId和password

2. 客户端用userId和password 计算得到一个签名：`H1=md5(password), S1'=md5(H1 + userId)`
3. 客户端构造一个消息：`message=randomKey;timestamp;H1;sigData`。其中randomKey是一个16位的随机数字；SigData为客户端的基本信息，比如userId, IP等。
4. 客户端用对称加密算法(推荐使用性能极高的TEA)对这个消息进行加密：`A1=E(S1', message)`，对称密钥就是上面的S1'，然后将userId和A1发送给服务端。

2. <<< 服务端

1. 接收客户端发送的userId和A1。
2. 根据userId，从用户信息表中得到用户注册时保存的签名`S1=md5(H1 + userId)`。注意服务器保存的不是H1。这个签名就是前面对称加密(TEA)的对称密钥。而且正常情况应该跟客户端根据用户输入的userId和password计算出来的S1'是一样的。
3. 服务端尝试用S1解密A1。如果解密异常，则提示登陆失败。如果解密成功，则按照约定格式，得到`message=randomKey;timestamp;H1;sigData`
4. 服务器对message中的用户信息进行验证
    1. 比较 `message.sigData.userId == DB.userId`; 如果不一样，那么很有可能数据曾被篡改，或者这是一个伪造的登录的请求，提示登录失败；
    2. 比较 message.timestamp与服务器当前的时间，如果差距较大，则拒绝登录，可以从很大程度上防止重放攻击。大家可能都有过经验，当本地时间与真实时间有较大差距的时候，总是会登录失败，其实就是服务器对客户端时间进行校验的原因。
    3. 比较 `md5(message.H1 + message.sigData.userId) == DB.S1`。如果不一致，则登陆失败。
5. 如果验证通过，说明用户身份认证通过，服务器同样构造了一个消息：serverMessage=whaterver you want，一般是登陆凭证，如sessionkey或者token. 然后用客户端发送过来的randanKey进行对称加密：A2=E(randanKey, serverMessage)，然后把A2返回给客户端。

3. 客户端 >>>

1. 客户端拿到A2，用randanKey进行解密，如果可以解密成功，则认为是来自真实服务器的数据，而不是一台伪造的服务器，整个登陆流程完成。这个步骤主要是为了防止服务器伪造。这个步骤很容易被忽略，客户端应该和服务器一样，要对接受的数据保持不信任的态度。

这个方案虽然没有使用HTTPS，但是思路跟HTTPS很类似。安全性也很高。

说明

1. 上面的认证过程其实就是著名的网络认证协议Kerberos的认证过程。Kerberos是一种计算机网络认证协议，它允许某实体在非安全网络环境下通信，向另一个实体以一种安全的方式证明自己的身份。它的设计主要针对客户-服务器模型，并提供了一系列交互认证——用户和服务器都能验证对方的身份。Kerberos协议可以保护网络实体免受窃听和重复攻击。QQ的登录协议都是基于Kerberos的思想而设计的。不过Kerberos也有一些安全性的问题。SRP协议Secure Remote Password protocol要更安全一些。据说是目前安全性最好的安全密码协议。
2. 为什么DB保存S1而不是H1。这是为了提高批量暴力破解的成本。对经过两次MD5加密，再加上userId作为salt，得到的S1进行暴露反推pasword基本是不可能的。而反推H1的话，由于H1的是password的MD5值，相对于password来说强度要增强不少。这个读者可以自己试试md5('123456')看得到的H1就有直观的认识了。
3. 为什么服务端解开对称加密后的message之后还要对message的内容进行验证。这是为了避免拖库后的伪造登录。假设被拖库了，黑客拿到用户的S1，可以伪造用户登录（因为我们的加密算法是公开的）。它能够通过服务器第一个验证，即服务端使用存储的S1能够解开消息；但是无法通过第二个验证，因为H1只能是伪造的，这是因为根据S1反推H1是很困难的（原因见上面Note.2）。
4. 但是虽然上面的协议可以防止黑客拖库后伪造用户登陆，却无法防止黑客拖库后伪造服务器，以及进行中间人攻击。拖库之后，黑客拥有DB.S1，协议又是公开的，这相当于黑客拥有了伪造Server的能力。黑客通过DB.S1解开客户端发送的A1，得到userId, H1=MD5(pwd), 以及randomKey。这样，黑客就可以轻易的得到被监听用户的H1了，而不需要通过S1进行反推。黑客就可以直接用H1伪造合法的用户登录请求了。如果pwd不够复杂，那么还很容易被暴力破解。黑客拦截客户请求，利用破解得出的randomKey回复一个假的响应，达到伪造Server的目的。黑客还可以同时伪造用户和Server，进行中间人攻击，从而达到窃听用户会话内容的目的。美国“棱镜门”就是类似案例。甚者在必要的时候对会话内容进行篡改。当然，黑客拖库后只能破解他监听网络所截取到的受害用户的MD5(pwd)，比起服务器存MD5(pwd)的影响面窄多了，所以建议服务器还是不要直接存MD5(pwd)。

可以在现有登陆流程基础上加上密钥交换算法（如ECDH）解决上面的问题。具体流程如下：

==前置条件==

1. 服务端生成一对私钥serverPrivateKey和公钥serverPublicKey，公钥直接hardcode在客户端，而不是通过网络传输。

1. 客户端 >>>

1. 用户输入userId和password
2. 客户端用对称加密算法对消息进行加密：`A1=TEA(S1', randomKey;timestamp;H1;userId)`，对称密钥 S1'=md5(md5(password) + userId)`
3. 客户端生成自己的一对公钥clientPublicKey和密钥clientPrivateKey。
4. 客户端用serverPublicKey对A1和clientPublicKey进行非对称加密: `ECDH(serverPublicKey, userId + A1 + clientPublicKey)`，然后将加密结果发送给服务端。

2. <<< 服务端

1. 使用serverPrivateKey对客户端发送信息进行解密，得到userId, A1 和 clientPublicKey。
2. 根据userId，得到DB.S1。然后尝试用DB.S1解密A1。如果解密异常，则提示登陆失败。如果解密成功，则按照约定格式，得到`message=randomKey;timestamp;H1;userId`
3. 服务器对message中的用户信息进行验证:
    1. 比较 `message.sigData.userId == DB.userId`
    2. 比较 message.timestamp与服务器当前的时间，如果差距较大，则拒绝登录，可以从很大程度上防止重放攻击
    3. 比较 `md5(message.H1 + message.sigData.userId) == DB.S1`
4. 如果验证通过，说明用户身份认证通过。服务器用客户端发送过来的randanKey进行对称加密：`A2=TEA(randanKey, sessionKey + 业务票据)`
5. 服务端使用clientPublicKey对A2进行加密: `ECDH(clientPublicKey, A2)`， 然后把加密结果返回给客户端。

3. 客户端 >>>

1. 客户端使用clientPrivateKey对服务器回包进行解密，得到A2。
2. 用randanKey对A2进行解密，如果可以解密成功，则认为是来自真实服务器的数据，而不是一台伪造的服务器，整个登陆流程完成。

4. ==后续通讯==

由于非对称加密算法(如RSA、ECC)的计算复杂度相比对称加密高3个数量级，不太可能用作通信数据的加密。所以验证通过之后，服务端和客户端利用ECDH交换密钥算法独自计算出一样的密钥，后面的通讯应该还是基于对称加密。这个跟HTTPS是类似的。

说明

加上非对称加密之后，即使黑客知道了S1，也没有办法伪造服务器，因为他不知道serverPrivateKey。黑客也不可能伪造客户端。因为即使黑客拖库知道S1，并且通过反编译客户端应用程序知道serverPublicKey，如果不是通过窃听网络得到A1，用S1解密A1从而得到A1中的H1的情况下，基本是不可能通过S1反推H1的。现在A1包在不对称加密算法上，黑客如果不知道serverPrivateKey和clientPrivateKey就无法解开会话信息。

这里看到serverPublickKey是HardCode在客户端的，而不是通过公钥推送下发的。这里是为了保证客户端的serverPublicKey是真正的serverPublicKey，而不是来自伪造服务器的。如果serverPublicKey通过网络传输下发给客户端的，那么伪造服务器在劫持网络的情况下，完全可以拦截客户请求，发回伪造服务器自己的*fake_serverPublickKey，客户端在不知情的情况就用*fake_serverPublickKey向伪造服务器发数据，因为*fake_serverPublicKey是黑客构造的，他当然有对应的*fake_serverPrivateKey解开。当然，黑客还需要知道S1才能解开A1。当然，将serverPublicKey硬编码在客户端会导致更换密钥需要强制客户端一起升级，否则老版本的客户端将无法登陆。

客户端自己也生成了公钥和私钥，目的是为了验证服务器是真正的服务器，这个得以成立的前提是公钥是真正的服务器公钥，如果是伪造服务器的公钥，那等于向黑客公开了自己加密的内容，黑客就知道了客户端的公钥，就能伪造服务器的返回数据。

采用ECDH非对称加密算法包裹后的登陆认证协议，安全级别已经跟SRP差不多。相对于SRP协议的问题是

ECDH安全性依赖于私钥的保密性，如果私钥泄漏安全性回到跟未加ECDH的时候一样。ECDH的私钥保密有以下一些办法（考虑到实现成本，推荐方案2）

1. 私钥加强保密，做到绝密，不用换，也不准备换；例如用USBKEY等硬件方式。
2. 私钥密码级别安全，定期更换，换的时候让客户端拉取。具体实现方法是第一步验密的成功的以后，发现要更新公钥，在返回的加密信道中返回新的公钥。
3. 使用Verisign等第三方证书，确保公钥的合法性，在2的基础上不会增加网络交互。
4. 完全使用TLS的协议，会导致每次登录增加一次交互。

另外，采用不对称加密算法，会有一个服务器间共享密钥的安全VASKEY的问题：

1. 换key困难，架构设计导致
2. 网络上对称加密传输新密钥，可能被破解

可能的解决方案：

1. VASKEY的业务Key管理目前是对称加密的方式，可以考虑换成ECDH或类似的密钥交换协议来动态更新Key。
2. 集中验证签名和解密，不共享密钥。

这里推荐采用方案2。

最后一个容易忽略的地方是注册。采用HTTP注册相当于明文传递用户名和密码，存在劫持、中间人攻击的危险。将注册入口全部改为HTTPS，封掉HTTP注册可以去掉上述危险。但是如果黑客劫持网络，给用户返回HTTP的页面，代理请求的HTTPS注册接口，也能窃取密码。

TIPS && THOUGHTS

1. 基本上所有的安全的登陆鉴权协议都是采用HTTPS的思想：使用不对称密钥算法进行登陆验证以及对称密钥交换。通过之后，后续会话就用对称加密算法加密。但是与HTTPS采用CA证书不同的是，一般通过密钥本身来验证客户端和服务器的合法性。
2. 对于对称加密算法安全性问题更好的解决办法是加密算法协商，而不在于寻找破解难度更大的算法。在验密码之前协商后续会话使用的对称加密算法，如果有安全性更好的算法或者正在使用的算法被破解了，只需要Server增加对更安全算法的支持，新版本的客户端就可以使用安全性更好的算法。支持对称加密算法协商以后可以随时动态的升级或替换加密算法，不用担心对称加密算法被破解的安全风险。HTTPS就是这样的一种思路。
3. HTTP登录无法防劫持，HTTPS登录可以防劫持，但是无法防止后续对业务会话的窃听。另外，CA也容易被伪造。
4. 关于暴力登录。处理方式有几种：1. 验证码； 2. 一定时间内密码错误次数超过正常值，锁定账户； 3. 客户IP监控。
5. 第三方爆库、钓鱼、扫号、撞库如果解决？这些问题等同于用户明文密码被盗，靠密码登录体系没有办法对抗，比较好的方式是对已知泄漏密码的用户进行封停，强制要求用户改密码等运营方式减少用户的财产损失。
6. 上面所有的验证方式，每次请求都需要根据userId去数据库拉取用户信息（password）。所以最好结合缓存进行处理，毕竟用户信息变化频率还是比较小的。

这种鉴权认证方式，相对于前面的token方式而言，客户端有比较多的计算和验证逻辑，比较适合于CS架构，特别是手机App。而且，它不依赖于Cookies，所以天然具有SSO功能。

More about 密码强度 && 暴力破解

1. 加盐之外我们还可以做些什么？

加盐以后的密码存储体系已经有效的对抗批量暴力破解、彩虹表攻击、字典攻击。为了对抗针对单个密码的暴力破解，我们可以通过密钥加强(Key Stretching)、提高密码长度等方式来提高安全性。

密钥加强的基本思路是让每次鉴权的时间复杂度大到刚好不影响用户体验，但是黑客暴力破解或构建彩虹表的成本大幅度提高，目前标准的算法有BCRYPT、SCRYPT、PBKDF2等。密钥加强算法的本质是通过加盐以及迭代计算多次增加计算量，迭代次数增加1000次计算量翻1000倍，这种方式对单个密码的暴力破解作用不大。

提高密码长度对构建彩虹表暴力破解的成本影响非常大，提高密码长度提升单个帐号的密码安全价值很大。以目前暴力破解MD5能力最强的FPGA NSA@home计算量来估算（大概用一台pc电脑的功耗，每秒钟可进行30亿次的8位密码（密码空间64个字符）尝试），假设有1万个NSA@home集成电路，19秒就能破解8字符的密码（70个字符的密码空间），所以现在的很多密钥检验要求的最小长度6个字符其实是不安全的，如果把密码长度增加到12个字符，则需要1年才能破解。

2. GPU，FPGA，ANIC对上述密码强度有何挑战？

FPGA MD5暴力破解工具目前比较成熟的是NSA@home，大概用一台pc电脑的功耗，每秒钟可进行30亿次的8位密码（密码空间64个字符）尝试。Lightning Hash Cracker[7]使用9800GX2 GPU每秒能够完成密码MD5计算608M次。

类似FPGA，GPU这种暴力破解方式受密码字符空间、密码长度、密码计算强度影响比较大，可以通过提升密码长度、密码计算强度来提升安全性。只替换密码Hash算法对这种破解作用不大，SHA2的计算量不到MD5计算量的一倍，把MD5换成SHA2只是让暴力破解的计算量增加了一倍，但是如果密码长度增加1位对计算量的提升是非常巨大的。

对于对称加密算法，没有专门针对TEA的硬件设计，针对AES算法，有一篇论文针对AES128专门设计的类似GPU的硬件计算速度能达到1012次/秒（约240次/s），AES128暴力破解需要288秒，这个计算量目前也是安全的。这种对称加密算法的暴力破解只能通过提升密钥长度来对抗。128位的AES与TEA安全级别差不多，AES支持最长256位密钥，长远来看对抗暴力破解AES更有优势。然而，对于对称加密算法安全性问题更好的解决办法是加密算法协商，而不在于寻找破解难度更大的算法。

3. 加密或者摘要算法的选择

1. MD5 VS SHA2

MD5不再具有抗冲突性，已经不适合用来做消息摘要、消息认证，但是MD5仍具有抗第一原像性（单向性），因此用MD5做密码存储是安全的。对用户密码做强度检验（特别是长度），并且在MD5的基础上加盐存储，可以大大提高对抗拖库后的批量暴力破解、彩虹表攻击、字典攻击能力。不过盐存放在哪里是个需要考虑的问题。

把MD5换成SHA2对现有密码存储体系安全等级没有质的提升，SHA2同样可以构造彩虹表。暴力破解SHA2的计算量不到MD5计算量的2倍。另外把MD5换成SHA2的工作量相当于再做一次密码加盐，但是对于安全等级没有质的提升，投入产出比不合理。如果系统已经使用了MD5加密，没有什么必要将其替换为SHA2。

2. AES VS TEA

从对抗暴力破解的能力来看。就目前的安全级别TEA是2^{126}，AES128的安全级别是$2^{126.1}$。128位的AES与TEA安全级别差不多，2^{126}的安全级别对于目前的计算能力来说是安全的。

对于硬件暴力破解对称加密算法，目前没有专门针对TEA的硬件设计。有一篇专门针对AES的FPGA暴力破解论文（《Cryptanalysis of the Full AES Using GPU-Like Special-Purpose Hardware》Biryukov, Alex and Großschädl, Johann. Fundamenta Informaticae 114 3-4, 221-237, 2012）。论文中为AES128专门设计的类似GPU的硬件计算速度能达到1012次/秒（约240次/s），AES128暴力破解需要288秒。TEA算法计算量跟AES差不多，这里用AES FPGA计算能力估算一下128位TEA的安全性：100万台上述计算能力的FPGA硬件暴力破解需要2.45万亿年，这个计算量是安全的。假设计算机单机能力18个月翻一番，则60年后，同样的设备数计算一年能够破解。这样看TEA是安全的。虽然TEA密钥长度只有为128位。但是它比DES要简单，加解密速度也比DES快很多，抗差分分析能力强，密钥空间大，安全性好。AES密钥长度有128位、192位和256位。长远来看对抗暴力破解AES更有优势。

3. RSA VS ECC

1024位的RSA已经不安全了。2010年，实现了对768位的大数因式分解。也许几年后1024位就被分解了。详细见RSA Factoring Challenge

ECC使用较短的操作数，就可以获得与RSA或离散对数系统同等的安全级别（大概为160~256位比1024~3072位），在性能和带宽上更有优势。

4. 蛮力攻击的大概门槛

使用蛮力攻击成功破解不同长度密钥的对称算法预计需要的时间：

密钥长度	安全评估性
56-64位	短期：只需要几个小时或几天破解
112-128位	长期：在量子计算机出现前，需要几十年破解
256位	长期：即使使用量子计算机，也需要几十年

需要强调的是：计算机能力在不断增强，按照摩尔定律，每15年计算机能力会增长1000倍，或者说每18个月，破解的成本降低一半。

Authorization

认证之后的下一步就是授权了。因为资源(体现在URL上)往往是有分访问等级的。有些是public的，匿名用户就可以访问。有些是私人的，必须登陆才行，有些是需要管理员才能处理的。所以我们还需要判断这个用户是否有权限对这个资源进行某些操作。这个是通过角色来处理的。

每个用户有个角色，每个资源也可以定义一个角色，就可以简单判断了。资源的访问权限配置可以通过全局的URL匹配，比如/my/xxx的URL必须用户登陆，/admin/xxx的必须是管理员权限。也可以是方法级别的细粒度，这个可以使用anotation简单的做到。

```
@Controller
@RequestMapping("/my")
public class MyController{

    @LoginRequired(role = { Role.ADMIN, Role.USER })
    public void doXXX(){
      xxx
    }
}
```

然后在Interceptor中进行处理。这个其实参考flask pricipal的。可惜Java不支持函数式编程，不能把函数做为参数传递。而在Python中，创建一个decorator是多么的简单Simple Authorization：

```
def requires_roles(*roles):
def wrapper(f):
    @wraps(f)
    def wrapped(*args, **kwargs):
        if get_current_user_role() not in roles:
            return error_response()
        return f(*args, **kwargs)
    return wrapped
return wrapper
```

然后就可以这样子使用了：

```
@app.route('/user')
@required_roles('admin', 'user')
def user_page(self):
    return "You've got permission to access this page."
```

发现Spring Security也是这么处理的。支持对URL进行匹配访问授权控制 1.6. Advanced Namespace Configuration:

```
<!-- Stateless RESTful service using Basic authentication -->
<http pattern="/restful/**" create-session="stateless">
  <intercept-url pattern='/**' access='ROLE_REMOTE' />
  <http-basic />
</http>

<!-- Empty filter chain for the login page -->
<http pattern="/login.htm*" security="none"/>

<!-- Additional filter chain for normal users, matching all other requests -->
```

```
<http>
  <intercept-url pattern='/**' access='ROLE_USER' />
  <form-login login-page='/login.htm' default-target-url="/home.htm"/>
  <logout />
</http>
```

URL还可以支持正则表达式匹配，只要声明一下path-type="regex":

```
<bean id="filterInvocationInterceptor"
      class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="runAsManager" ref="runAsManager"/>
  <property name="securityMetadataSource">
    <security:filter-security-metadata-source path-type="regex">
      <security:intercept-url pattern="\A/secure/super/.*\Z" access="ROLE_WE_DONT_HAVE"/>
      <security:intercept-url pattern="\A/secure/.*\" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
    </security:filter-security-metadata-source>
  </property>
</bean>
```

还可以支持Spring EL表达式：

```
<http use-expressions="true">
    <intercept-url pattern="/admin*"
        access="hasRole('admin') and hasIpAddress('192.168.1.0/24')"/>
    ...
</http>
```

也可以是方面粒度的访问控制：3.7. Method Security

```
public interface BankService {

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();

  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);
}
```

注解同样支持Spring EL表达式：

```
@PreAuthorize("#contact.name == authentication.name")
public void doSomething(Contact contact);
```

Spring-Security建议在Service接口上做注解保护。在Controller类上做注解保护其实基本也可以达到URL匹配目的，结合Spring MVC的interceptor机制，不失为一个快速简洁的实现。

TIPS

1、将用户身份存放在ThreadLocal中可以方便后面的权限判断。Spring security提供了两种方式获取这个认证身份：

法一：通过`Authentication authentication = SecurityContextHolder.getContext().getAuthentication();`获取

```
import org.springframework.security.web.bind.annotation.AuthenticationPrincipal;

// ...

@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser() {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
```

```
    CustomUser custom = (CustomUser) authentication == null ? null : authentication.getPrincipal();

    // .. find messags for this user and return them ...
}
```

法二: 通过`@AuthenticationPrincipal`注解进行参数注入 (要求Spring Security 3.2+)

```
@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@AuthenticationPrincipal CustomUser customUser) {

    // .. find messags for this user and return them ...
}
```

内部是通过`AuthenticationPrincipalArgumentResolver`实现的，具体参见: 11.2. @AuthenticationPrincipal。

也可以通过Spring的queust scope bean达到同样的效果。具体参见笔者前面的文章Spring的Bean Scopes。

2、Spring Security提供了一些加密相关的工具类和方法，可以参考使用：9. Spring Security Crypto Module。

参考文章

1. 你必须了解的Session的本质
2. 你会做Web上的用户登录功能吗？
3. The definitive guide to forms based website authentication 这篇帖子真是有意思，回答基本都是长篇博文了[/偷笑]
4. Building and implementing a Single Sign-On solution 关于单点登陆笔者目前看到最好的文章了。
5. Remember me in Spring Security 3 Spring Security的Remember Me实现介绍，图文并茂，推荐阅读。
6. Elliptic curve Diffie–Hellman
7. Elliptic curve cryptography
8. MD5
9. SHA-2
10. Key stretching
11. NSA@home
12. 新浪微博的XSS攻击
13. Shiro Security

Related Posts

- 24 Jul 2015 » Tomcat调优
- 22 Jul 2015 » 记一次MySQL主从同步错误处理
- 03 Jul 2015 » Metric监控系统