Implementing a one-time password system on the web

DOUBLE PROTECTION

Add security to your website with a one-time password system. **BY JAMES A. BARKLEY**

Two-factor authentication is a system in which two different factors are used in combination to authenticate a user. Two factors, as opposed to one factor, will deliver a higher level of authentication assurance. The combined factors could consist of:

- Something the user knows (password or pin)
- Something the user possesses (smartcard, PKI certificates, RSA SecurID)
- Something the user is or does (fingerprint, DNA sequence)

The first option is the easy choice. Passwords are used everyday for a multitude of purposes. The third option is usually some sort of biometric – not a good choice for the web environment. "Something the user possesses" is the best second factor for authentication. Almost all web-based, two-factor authentication solutions available today involve some form of hardware token, such as the RSA SecurID. Distributing these tokens to users is neither cost effective nor scalable in price. A company might be able to afford tokens for 1,000 users, but one good blog post and they could find themselves with 30,000 new users overnight. Requiring users to obtain a hardware token on their own is too much work for the vast majority of users. In addition, tokens have to be synced with special server software, which can often require a proprietary license.

A less expensive and more scalable alternative for two-factor authentication on the web is a one-time password (OTP) system. The November 2008 issue of *Linux Magazine* offered an introduction to OTPs [1] that focused primarily on workstation authentication; however, tasks like checking a bank account from an untrusted network scream for some form of two-factor authentication, and an OTP system is often a practical solution. In this article, I describe how to add the security of OTPs to your website.

OTP on the Web

RFC 2289 [2] defines an OTP system derived from Bellcore S/KEY technology (RFC 1760). If implemented correctly, it provides a cost-effective, two-factor authentication solution for websites. Imagine that a help desk technician with administrative privileges for a website hits an administrative page that generates a wallet-sized list of 30 OTP number/key pairs. The list is then hand delivered to the user. This password list now becomes something the user possesses – the second factor – and because it was never transmitted electronically, it provides an added level of security. If the site doesn't mind electronic transmission within its trusted domains, the admin might fax or even email the list to the user. From a cafe in Amsterdam, for example, the user can now enter a conventional username and password. If this initial authentication is successful, the server poses a challenge that requires a response with the correct corresponding OTP. After this login, the OTP is immediately invalidated for future use, which means it will never be used for a replay attack. For the next login, the user will enter the next OTP on the password list.

By forcing the user to authenticate through a pair of dissimilar mechanisms,

two-factor authentication provides a much more secure alternative for web login. This basic scenario leads to endless variations. For instance, a user could associate a cell phone number with the account; then, when logging in, the system could send the OTP in a text message. Or, a user could generate OTP passwords from a program running on a PDA. An added advantage of this scenario is that the implementation can give the OTP a temporal component so that it times out after 60 seconds, much like the RSA SecurID, although this would require the user synchronizing the PDA application with the server.

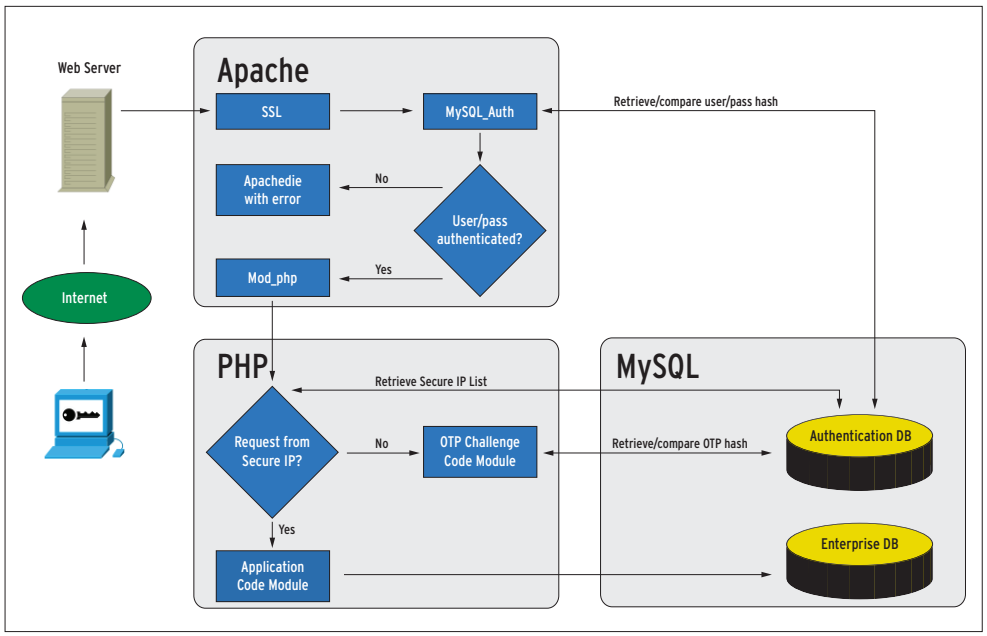

Figure 1: An OTP web login scenario.

OTP Tools

A plethora of OTP libraries exist for SSH, console, and network logins, with plenty of OTP libraries for more exotic tools like SquirrelMail and PalmPilots, but finding open source libraries for OTP authentication in a web environment is difficult. One RFC 2289–compliant OTP system that has been tested and released under the GPL is the OTPauth PHP library [3]. OTPauth, which uses the SHA1 hashing algorithm, has been employed successfully by a site with several hundred users for over two years. Another PHP library, otp, is available from SourceForge [4]. The otp developers hope to have a demo up and running soon.

Various Java tools assist with the task of constructing and validating OTPs [5], but I am not aware of a complete web library (e.g., something that integrates a full challenge/response implementation into a j2ee application).

The Google AuthSub library [6] allows authentication with Google applications

Listing 1: Example Authentication Database

```
01 CREATE TABLE user (
02   user_id int(11) NOT NULL AUTO_INCREMENT,
03   user_name text NOT NULL,
04   user_pw varchar(32) NOT NULL DEFAULT '',
05   realname varchar(32) NOT NULL DEFAULT '',
06   STATUS char(1) NOT NULL DEFAULT 'A',
07   add_date int(11) NOT NULL DEFAULT '0',
08   confirm_hash varchar(32) DEFAULT NULL,
09   phone_number varchar(20) NOT NULL DEFAULT '',
10   last_pw_change int(11) NOT NULL DEFAULT '0',
11   otp_enabled tinyint(1) NOT NULL DEFAULT '0',
12   PRIMARY KEY  (user_id),
13 ) TYPE=MyISAM;
14
15 CREATE TABLE session (
16   user_id int(11) NOT NULL DEFAULT '0',
17   session_hash char(32) NOT NULL DEFAULT '',
18   ip_addr char(15) NOT NULL DEFAULT '',
19   otp_auth tinyint(1) NOT NULL DEFAULT '0',
20   time int(11) NOT NULL DEFAULT '0',
21   locked tinyint(1) NOT NULL DEFAULT '0',
22   PRIMARY KEY  (session_hash),
23 ) TYPE=MyISAM;
24
25 CREATE TABLE otp (
26   user_id int(11) NOT NULL DEFAULT '0',
27   sequence int(11) NOT NULL DEFAULT '0',
28   otp char(60) NOT NULL DEFAULT '',
29   PRIMARY KEY  (session_hash),
30 ) TYPE=MyISAM;
```

Listing 2: Adding mod_auth_mysql to httpd.conf

```
01 #loads mod_auth_mysql shared object library
02 LoadModule mysql_auth_module modules/mod_auth_mysql.so
03
04 #tells mod_auth_mysql how to connect to your auth database,
   parameters are:<br>#Auth_MySQL_Info hostname user password
05 Auth_MySQL_Info localhost auth_db_user myP@ssw0rd<br>
06
07 ?redirects failed logins to rejection page
08 ErrorDocument 401 /chapter14/rejection.html
09
10 #defines MySQL tables and columns for authenticating
11 AuthName "My Web Site"
12 AuthType Basic
13 Auth_MySQL_DB auth_db
14 Auth_MySQL_Encryption_Types MySQL
15 Auth_MySQL_Password_Table user
16 Auth_MySQL_Username_Field user_name
17 Auth_MySQL_Password_Field user_pw
18
19 require valid-user
```

via secure tokens. Although AuthSub is not a strictly RFC 2289–compliant OTP solution, it does allow secure, one-time token-style authentication for Google applications. It will be interesting to see whether Google continues to develop this solution or migrates completely to OAuth. A handful of other software packages provides a customized OTP solution exclusively for their software, such as a plugin for the Joomla CMS [7].

Here, I describe how to set up an OTP system with the open source OTPauth library. The other tools operate on similar principles. If you are interested in exploring one of the alternatives, see the documentation at the project website.

A Look at Self-Service OTP

Imagine a bank that wants to encourage good security practices but cannot insist on universal two-factor authentication without scaring away half of its customers. The bank wants a system that supports the OTP option for early adopters without endangering the business model by forcing constraints on the unwilling.

The solution must provide the means for a user to visit a preferences page and specify that the program require two-factor authentication when logging on to the account from a computer other than that currently being used. The user then generates a personal, wallet-sized list of 30 OTP number/key pairs from the user preferences page. The next time the user accesses the account from an untrusted location, say a friend's house, the user will be asked to provide an OTP along with the conventional username and password.

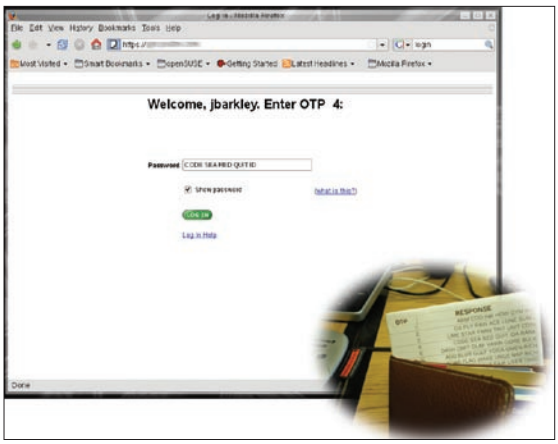

Figure 2: Logging in with a one-time password.

The first step is to provide basic authentication with a username and password. Great libraries and standard methodologies provide basic authentication, whether you want to use *.htaccess* files with Apache, validate off of a database at the application layer, or let Apache validate the user with *mod_auth_mysql*. Because I like to provide security controls at multiple layers, I will use the architecture shown in Figure 1.

The authentication database is stored separately from the enterprise database and holds the username, password, and OTP information. Listing 1 shows MySQL *CREATE* statements that contain

Listing 3: PHP OTP Logic

```
01 <?php
02
03 ...
04 ...
05
06 //retrieve user id from global set by Apache
07 //or similar mechanism
08 $uid = user_getid();
09
10 //setup user session
11 $session = user_getsession($uid); //attempt to retrieve
   session from db
12 if (!$session) {
13   $session = user_create_session($uid); //create entry in
   session table
14 }
15
16 //check to see if user is already authenticating
17 //this prevents RFC 2289 specified race condition
18 while ($session['locked']) {
19   /* spin until lock is released or timeout happens */
20   $session = user_getsession($uid);
21   if (spinlock_timeout_reached()) {
22       header("Location: http://www.example.com/retry.php");
23       exit;
24   }
25 }
26
27 //lock account while authenticating
28 set_session_lock($uid); //sets "locked" flag on session
   table
29
30 //check if otp auth has been enabled on account
31 //user_getotpauth() performs database query and returns
32 //otp_enabled flag from user table
33 $otp_auth_enabled = user_getotpauth($uid);
34
35 if ($otp_auth_enabled) {
36   if ($session['otp_auth']) {
37     /* success, user has already authenticated with otp */
38   } else {
39     /* user has logged in but not otp auth'd */
40
41     //untrusted_host() compares the IP of the current
42     //session with the user's specified trusted list
43     if (trusted_host($uid)) {
44       /* user is coming from address which won't require OTP
   auth */
45     } else {
46       /* user must otp auth */
47       header("Location: http://www.example.com/otp_auth.
   php");
48       exit;
49     }
50   }
51 }
52
53 //in all but otp required case
54 //the user ends up here
55 //release lock and proceed to page-specific code
56 unset_session_lock($uid);
57
58 ...
59 ...
60
61 ?>
```

Listing 4: Sample OTP Authentication Page

```
01 <?php
02 /* LICENSED UNDER THE GPL */
03 # if they have clicked the login button
04 if ($login) {
05    $success = valid_otp($form_challenge_response, 06$user);
06    if ($success) {
07       /* update session/auth state and redirect to system
         resources and exit*/
08       header("Location: http://www.example.com/".$page);
09       exit();
10    }
11 }
12
13 $sequence = get_otp_seq($uid);
14 if ($sequence == -1) {
15     /* print error message and exit; */
16 }
17
18 print "
19     <p>
20     <FORM ACTION=\"$PHP_SELF\" METHOD=\"POST\">
21     <p>
22     <INPUT TYPE=\"TEXT\" NAME=\"user\" VALUE=\"$user\">
23     <p>
24     Enter One-Time Password for Challenge number
       <B>$sequence</B>:
25     <br><INPUT TYPE=\"TEXT\" NAME=\
       "form_challenge_response\" VALUE=\
       "$form_challenge_response\" SIZE=\"31\">
26     <p>
27     <INPUT TYPE=\"SUBMIT\" NAME=\"LOGIN\" VALUE=\"Login\">
28     </FORM>
29     <P>
30     ";
31 ?>
```

all the information needed for the entire authentication database, which *mod_auth_mysql* checks for the username and password. First you need to download *mod_auth_mysql* or get it from your package manager [8]. Once you have it installed, configure it by adding the lines from Listing 2 to your *httpd.conf* file. Remember to customize the settings for your website.

Now you can add code at the application layer to check for OTP authentication. The user will never get to the application without entering the correct username and password, but once done, you need to make sure a two-factor authentication option is available. First, the application must determine whether the user has enabled OTP authentication for the account. If so, the application needs to compare the current IP address and/or hostname with those listed on the user account as trusted. If the user has not enabled OTP authentication or is coming from a trusted address, then the application allows access to the web page(s) requested. The sample code in Listing 3 could be *include*d in your application pages at the top (in which case, it is executed first) or moved to a function.

When the user is redirected to the *otp_auth* site, your OTP library handles the OTP challenge/response. Listing 4 shows a basic page that presents the challenge to the user and validates the response to the challenge. This page is intentionally sparse because my application is not yet convinced that this person is authentic. By not providing all my normal libraries or JavaScript, or even the look and feel of my site, I narrow the attack vectors on this page. The use of a good OTP library simplifies this application logic to a trivial amount of code with function calls like *valid_otp()* and *get_otp_seq()*. The code in Listing 4 produces a display similar to that shown in Figure 2.

Finally, don't forget to provide your

Risk Assessment and Attack Vectors

The biggest risk with any security system would be implementing an architecture coding mistake. The technical risk is getting hacked; the business risk is getting sued. To protect yourself, you could write extensive test cases and have a third party review the software, or you could choose an existing implementation proven or peer-accepted as correct.

Architecturally, secure programming requires a number of considerations, many unique to the existing infrastructure of the system. The two most important are: 1) separate your authentication database from your enterprise database(s) and 2) separate the username/password authentication from the OTP authentication as much as possible. You want each authentication element to be clean and almost stand alone. If you handle username and password authentication in the same code library as the OTP authentication, you risk contaminating the entire authentication process with cross-code bugs.

Another technical vulnerability in any OTP system is the man-in-the-middle (MITM) attack vector. If hackers can masquerade as a website to the user while masquerading as a user to the website, there is very little they cannot do. Even with SSL, the MITM simply makes separate SSL connections and decides what to send to user and website. Network monitoring can prevent this type of attack: As the attacker continues to operate, patterns of errors emerge.

RFC 2289 allows hashing algorithm variations in OTP generation and authorization routines. Be sure that a strong hashing algorithm is in use because an intruder that gains access to your authorization database and hashes the stored tokens with an insecure algorithm such as MD5 could generate an infinite list of tokens. At least, you should use SHA1, although it is not highly secure. SHA256 is a better solution but is not available in as many languages, which means relying on and validating a third-party library or writing your own. The RFC also mentions an interesting race condition (see the "Race Condition" box), so make sure your library is fully compliant.

Another concern is social engineering the help desk staff or users. Sadly, most solutions to that problem are non-technical. Most systems with help desk staff are susceptible, including commercial hardware token-based authentication systems, so good models to establish identity already exist. Training or a Terms of Service (TOS) agreement might be advantageous. In practice, most places with enough security requirements to warrant the use of OTP already require yearly online security training. This training or TOS also should prohibit the storage of OTP lists with regular usernames and passwords.

Listing 5: Spreadsheet for Generating OTP List

```
01 $otp_list = generator($uid);
02 /*
03     pretend we're an excel file and let
       excel or oo.calc put the html table cells
       into the right spreadsheet format
04 */
05 header("Content-Type:  application/vnd.ms-excel");
06 header("Expires: 0");
07 header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
08
09
10 print "<TABLE BORDER=1>";
11 print "<TH>Sequence number</TH><TH>Password</TH>";
12 while (list($key, $val) = each($otp_list)) {
13         print "<TR><TD>$key</TD><TD>$val</TD></TR>";
14 }
15 print "</TABLE>";
```

users with tools for enabling OTP on their accounts, generating their OTPs, and managing their trusted lists. Listing 5 is for a very lightweight page to generate a spreadsheet of OTPs, but make sure that when a user enables OTP for an account, it isn't logged out before generating the OTP list first!

Be prepared to have a mechanism for resetting OTP lists. This could be a responsive phone/email/irc support channel or an automated page, but either way, the user will need to provide proof of identity with something else like a security question. Also, don't forget your additional security checks – none of the code samples listed here validate input data, for example.

Not a Token

The RFC 2289 specification for a one-time password solution can offer true two-factor authentication; however, it will never be as secure as a token-based alternative. For one thing, many of the token-based solutions require that you concatenate a private PIN to the OTP to create the second factor, which greatly enhances security. Also, the hardware token solutions are designed to be tamper proof, in case someone tries to reverse engineer the generating algorithm. Finally, the token-based tools are time based and change every minute or so, which means it is very difficult for an attacker to obtain an OTP a user has not yet used. With solutions that require an OTP list, an attacker who gets a snapshot of the list (or picks up a lost list on the subway) has access to future OTP responses.

The OTP system defined by RFC 2289 offers an open and scalable solution for web-based authentication. It is even possible to integrate an OTP system into a user's cell phone. Web-based OTP has its own attack vectors and risks, and a web-based OTP system will probably never be quite as secure as hardware-based solutions such as the RSA SecurID. Despite this, OTP combined with a conventional web authentication scheme is an excellent candidate for poor man's two-factor authentication. ■

Bitwise

Implementation of the RFC 2289 specification used in OTPauth was written for PHP4 and also works with PHP5. To implement the spec correctly, a variety of bitwise operations are necessary. However, at the time of implementation (and I don't think it has changed), specific bitwise operations do not work in PHP4. Things like bit shifting for unsigned 32-bit integers don't work. PHP4 provides the operator, but it simply fails with no error. Therefore, OTPauth provides a math library to work around these types of undocumented language "features."

Race Condition

One race condition exists for the OTP system. An attacker who is listening with a keystroke logger while a user is authenticating might be able to listen to just enough of the OTP to enable a brute force attack just before the user finishes typing, allowing the attacker to log on as the user before the user finishes authenticating the session. Interestingly, RFC 2289 actually has a provision for this race condition and requires that it be guarded against in order for the implementation to claim full compliance. The defense outlined in Section 9.0 of the RFC is to deny multiple simultaneous sessions. In other words, once a user initiates the authentication sequence, all other attempts to authenticate with that user should be blocked until the authentication process is complete. This could lead to a denial of service attack, so some sort of authentication timeout is necessary.

INFO

[1] "Smart Access" by Udo Seidel, *Linux Magazine*, November 2008

[2] RFC 2289: *http://www.apps.ietf.org/rfc/rfc2289.html*

[3] OTPauth: *http://code.google.com/p/otpauth/*

[4] otp: *http://sourceforge.net/projects/otp/*

[5] OTPs in Java: *http://www.java2s.com/Code/Java/Security/OTPonetimepasswordcalculation.htm*

[6] Google AuthSub library: *http://code.google.com/apis/accounts/docs/AuthSub.html*

[7] Joomla OTP plugin: *http://code.google.com/p/joomla-otp-auth/*

[8] Apache *mod_auth_mysql* module: *http://modauthmysql.sourceforge.net/*

[9] Phishing attack on OTPs: *http://www.theregister.co.uk/2005/10/12/outlaw_phishing/*

THE AUTHOR

James A. Barkley is a Senior Software Systems Engineer at the MITRE Corporation in Bedford, MA. Mr. Barkley is working in a variety of technology areas including Modeling and Simulation, PHP and Ruby on Rails web development, biometric informatics, server virtualization, and robotics. MITRE is a not-for-profit company that manages three federally funded research and development centers (FFRDCs) and a dedicated homeland security center, partnering with government sponsors to support their crucial operational missions. Mr. Barkley began working with open source software in 1998. He is the creator of the OTPauth PHP library.