

Further Web Security Enhancements

Joshua Small

www.lolware.net

Intro

Several years ago, documents started surfacing featuring the now famous SQL injection issues and similar. I've recently worked with a series of web developers and found that, in general, the message HAS sunk in. People are parameterising their code, and following the basics. Whilst there is still an unfortunate amount of "I'm not a big company and can't have a security issue so I'm not going to bother writing one line of code differently even if it means my application crashes when someone has a ' in their name" still going on, it's time we let those people lie in the bed they've made and offer something new. I've done some further research lately, and while there are plenty of new web security documents being written, I don't see anything that wasn't bought up five or more years ago.

This document details some further enhancements I have been utilising. Nothing here should be considered a substitute for good code writing. But we chroot BIND, we avoid wuFTP and patch daemons that are firewalled off from the outside world and we refuse to run anything as the root user. We don't do this because we know there are open holes in these applications (although given the history of some, it's a matter of time). We do it as a harm minimisation technique, and I'd encourage the reader to consider further harm minimisation with the advice below.

To demonstrate these techniques, I have developed a woefully terrible database interface. Although you aren't expected to produce anything like this in the real world, you can see from its example how a compromised web site could be used to complete the commands demonstrated. The associated tables and dbwork.pl web interface can be found in the appendix. I urge you try not to fall back into the argument that "it won't be necessary if you write good code". We've been there for years and exploitable websites just keep popping up. After all, you have to be realistic about the just how well written something developed by a single developer working after hours on a website can really be expected to be.

Passwords

Everyone has read all about placing SQL passwords in a secure location outside the webroot and not as a variable inside your main script. A simple demonstration of this occurs here:

DBWORK.PL:

```
use vars qw( $pass );  
require '/u01/sqlpass.pm';
```

```
..
```

SQLPASS.PM:

```
$pass = 'SQLPASS';  
1;
```

What's wrong with that? Look what happens when our buggy include script gets exploited. The first thing that occurs is, they see the above pathname. This becomes the output when accessing the dbwork.pl file appropriately:

```
Output for SQL Command select * from users was  
Row: 1 bill he likes men pass  
Row: 2 ted dude pass  
File /u01/sqlpass.pm contents are: $pass = "SQLPASS"; 1;
```

So while the code is "hardened", I see include bugs like this all the time, and the webroot has little effect on them. Not great! An improvement on this is the below version.

DBWORK.PL:

```
use vars qw( $pass );  
require '/u01/sqlpass.pm';  
  
$dbh = DBI->connect('dbi:Oracle:host=localhost;sid=mumlol',  
    'websql', ` $pass`, {  
    RaiseError => 1,  
    AutoCommit => 0  
    }  
)  
|| die "Database connection not made: $DBI::errstr";  
$pass = undef; #Scrub the password
```

Note the backticks! Also importantly, note we remove it from memory as soon as it's used. This way there can be no "code exposure" exploits revealing the contents of the variable

SQLPASS.PM:

```
$pass = "/usr/local/bin/sqlpass unlockcode";  
1;
```

So what we can see is that an outside binary application is called to produce the password to the Perl script. The application itself also features an "unlock code" so it won't just spit out the password to anyone who asks.

SQLPASS.C

```
#include <stdio.h>  
#include <string.h>  
#define UNLOCK "unlockcode"  
  
int main(int argc, char**argv)  
{  
    if(!argv[1] || (strcmp(argv[1],UNLOCK) != 0))  
    {  
        printf("No code for you!\n");  
    }  
}
```

```
        return 0;
    }
    printf("SQLPASS");
    return 0;
}
```

When a user attempts to “includefile” the sqlpass binary, the browser does its best to display the output as text, and produces a largely garbled screen. The password IS there, but with something less obvious than “SQLPASS” it’s still likely to go unnoticed. An exercise for the reader is to expand on this, storing the password as an XOR of two hex strings. Since the unprintable characters never make it to your screen, recovery should be quite difficult.

User Details

Passwords are an obvious one. Everyone already knows how to apply crypt(). Below you can see my logon database searched from my exploitable website, and we can imagine that the crypt() technique has already been used:

```
Output for SQL Command select * from users was  
Row: 1 bill he likes men cryptpass  
Row: 2 ted dude cryptpass
```

Many web developers would say this is great, as the pass is crypt()ed and therefore “secure”. Aside from the fact that such hashes ARE (albeit, often very slowly) attackable (very easily for obvious passwords), I’m sure Bill would disagree with his private_info field being accessible, particularly considering its content. Your marketing department will also love you forever if you can use the “E” word.

Encryption is the answer, but many (let’s just say it, MSSQL developers) have failed at this big time by having the application perform the encrypt/decrypt, with a global key stored right there in the application.

The better way, is to store the password in the database- not in a table, but in a procedure which isolates the user from the encryption process. What use is it if the user gets his code automatically encrypted and decrypted? It’s plenty of use – when you define exactly what decryption process is required and don’t create anything else.

Oracle’s encryption toolkit is discussed here:

<http://www.oracle-base.com/articles/8i/DataEncryption.php>

Only we are not going to create the decryption function. What the users will need is:

1. The ability to enter data into the table
2. The ability to confirm a password (check a logon)
3. To be able to recover the private_info field, but this is only necessary for the current user.

We achieve this as follows:

1. Alter our table such that “pass” and “personal_information” use the RAW data type.
2. Create the package above, and associated functions, without the “decrypt” function.

3. Create a “userlogon” function as below.

Review the appendix for the current table layout.

```
CREATE OR REPLACE PACKAGE toolkit AS
  FUNCTION encrypt (p_text IN VARCHAR2) RETURN RAW;
  FUNCTION check_logon (username IN VARCHAR2, inpass IN VARCHAR2) return VARCHAR2;
END TOOLKIT;
/
```

.. encrypt and padstring are defined in Oracle’s document. Be sure to change the global key to something secret.

Here is my custom written check_logon function:

```
FUNCTION check_logon (username IN VARCHAR2, inpass IN VARCHAR2) return VARCHAR2 IS
  l_decrypted VARCHAR2(32767);
  pass_raw RAW(128);
  pers_raw RAW(512);
  personal VARCHAR2(512);

BEGIN
  SELECT pass,private_info INTO pass_raw,pers_raw FROM users
  WHERE name = username;
  DBMS_OBFUSCATION_TOOLKIT.desdecrypt(input => pass_raw,
    key => g_key,
    decrypted_data => l_decrypted);

  l_decrypted := RTrim(UTL_RAW.cast_to_varchar2(l_decrypted), g_pad_chr);

  IF l_decrypted != inpass THEN
    RETURN 'INVALID';
  END IF;
  DBMS_OBFUSCATION_TOOLKIT.desdecrypt(input => pers_raw,
    key => g_key,
    decrypted_data => personal);

  personal := RTrim(UTL_RAW.cast_to_varchar2(personal), g_pad_chr);
  RETURN personal;
END;
```

What is exciting about this function is that it achieves several security goals at once. We secure our passwords, and we also make it impossible to acquire the private_info without first identifying ourselves.

Now let’s help make the input process a little more transparent:

```
CREATE OR REPLACE TRIGGER encrypt_users
BEFORE INSERT OR UPDATE ON users
FOR EACH ROW
```

```

DECLARE
BEGIN
  :new.pass := toolkit.encrypt(UTL_RAW.cast_to_varchar2(:new.pass));
  :new.private_info := toolkit.encrypt(UTL_RAW.cast_to_varchar2(:new.private_info));
END;
/

```

Add some users (unfortunately the CAST is required in this example, feel free to functionalise the input too):

```

SQL> INSERT INTO USERS (name,pass,private_info) VALUES
('john',UTL_RAW.cast_to_raw('cryptpass'),UTL_RAW.cast_to_raw('big secret'))
SQL> INSERT INTO USERS (name,pass,private_info) VALUES
('kevin', UTL_RAW.cast_to_raw('cryptpass'),UTL_RAW.cast_to_raw('bigger secret'));

```

Let's watch a hacker executes the same private_info exposure hack he did earlier:

```

Output for SQL Command select * from users was
Row: 3 7E35E9E9E8EB8D83F2FC79EBC98BACDF john 374DA3DF45E0C72B9795B2433218517B
Row: 4 7E35E9E9E8EB8D83F2FC79EBC98BACDF kevin AD59C90353503078C8A1BF42D9BFEF0C

```

Quite useless. Note also you cannot really brute force the passwords. Without knowing the plaintext password, your application has no way of knowing if it's successfully cracked the code or not. And thanks to the fact I will still tell you to crypt() your passwords, which uses a salt, a hacker cannot just work by attempting to crack his own, known password first, because he won't know the crypt()ed version.

I have executed these directly from SQL*Plus to save myself adapting the Perl script, because function calls require a slightly different interface. But you can see that taking a username/password input, you would call toolkit.check_logon(name,pass) and the output will be either INVALID or contain the user's private info, to demonstrate a successful logon.

```

SQL> BEGIN
  2 DBMS_OUTPUT.put_line('Function called with correct pass: ' ||
  3 toolkit.check_logon('john', 'cryptpass'));
  4 DBMS_OUTPUT.put_line('John is a hakka! ' ||
  5 toolkit.check_logon('john', 'lolpass'));
  6 END;
  7 /
Function called with correct pass: big secret
John is a hakka! INVALID

```

Also note it fails on a non existent user:

```

SQL> BEGIN
  2 DBMS_OUTPUT.put_line('Not a real user' ||

```

```
3 toolkit.check_logon('arhg', 'password');
4 END;
5 /
BEGIN
*
```

```
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "WEBSQL.TOOLKIT", line 32
ORA-06512: at line 2
```

You can either trap this in your Perl code and deal with it there, or check the username in a prior with another SQL statement command.

Note the user names are not encrypted – it would just make life hard. Besides, you’re likely to want to search based upon the usernames at some point. And definitely don’t encrypt the ID.

A little note about where the encryption key is now. The “DESC functionname” command is an SQL*Plus command, not an SQL command, and it simply won’t work from DBI. This means simply “print me out the encryption function” is easier said than done. I have seen various workarounds, but they are quite fiddly and I couldn’t get them to work when doing it deliberately. The odds of having the unusual application code required to actually print the output of the functions involved are next to nil. Still, if you are concerned, you could always look into the next step of inserting the function into the database as Java bytecode.

Error pages

The standard advice is “don’t display errors to the end user”. But I’ve just passed my demo page some bad hacker data and this is all that got logged:

```
[Wed Jun 18 14:31:24 2008] [error] [client 192.168.1.100] [Wed Jun 18 14:31:24 2008]
dbwork.pl: Failed SQL with: ORA-01756: quoted string not properly terminated (DBD ERROR:
OCISstmtPrepare) at /var/www/cgi-bin/dbwork.pl line 37., referer:
http://192.168.1.105/index.html
```

This may tell me what’s wrong with my code – it may not. It’s particularly useless in telling me what variable my hacker was playing with. Here I would suggest looking further into CGI::Carp.

This is a bigger issue than it sounds. Very few exploits work first try and you’ll usually see half an hour of a hacker playing before he achieves something. It’s a matter of whether you log enough to find the hole he’s working with and close it.

A great solution was to add this to the start of the dbwork.pl file. You would likely expand on the aesthetic aspect of the output – but try not to call any other modules, or do anything potentially as buggy as the code you’re protecting. An “or die” here would be somewhat hopeless.

```
use CGI::Carp qw(set_die_handler);
BEGIN {
sub handle_errors {
my $msg = shift;
print "<h1>Error Page</h1>";
print "<p>Sorry, this page has died. I lol at your face.</p>";
```

```

open FH, '>>/tmp/perlcrash.log' or
    exit; #Die won't be much use here

print FH "-----\n";
print FH "Application $0 failed epically at " . localtime() . "\n";
print FH "Error was $msg\n";
print FH "Given variables were:\n";
foreach $key (param)
{
    print FH "$key was " . param($key) . "\n";
}

close FH;
exit;
}
set_die_handler(\&handle_errors);
}

```

Note with CentOS 5.1 I had to upgrade CGI::Carp with CPAN to get support for this code.

And I've just been greeted with this:

```

-----
Application /var/www/cgi-bin/dbwork.pl failed epically at Wed Jun 18 15:00:39 2008
Error was [Wed Jun 18 15:00:39 2008] dbwork.pl: Unable to open file: No such file or directory at
/var/www/cgi-bin/dbwork.pl line 77.
Given variables were:
sql was select * from users
includefile was /etc/passwlol

```

Here I can see that someone is using the includefile variable to open files, and fortunately didn't get it right the first time. Now I know where the hackers are working and where the flaws are in my code. Just be sure to actually read the file regularly!

Apache Modules

There are 52 modules loaded into Apache on a default CentOS install. Observe this snippet from httpd.conf:

```

LoadModule ldap_module modules/mod_ldap.so
LoadModule authnz_ldap_module modules/mod_authnz_ldap.so

```

Are you using LDAP in any way from your server? Probably not. Go down the list and comment out what you might not need. Be methodical and test thoroughly to ensure you didn't break anything. Remember the exploit in mod_rewrite? You wouldn't have been so concerned if you commented this line out:

```

LoadModule rewrite_module modules/mod_rewrite.so

```

And since you're probably not doing any reverse proxying, there are six related modules right there. That's a lot of code you can avoid loading, and a lot of potentially exploitable code you can negate.

Mod_security

Again, this has been touched on by others, so I won't discuss it at length. But this is too rarely used, so please reconsider implementing it. There's plenty of documentation out there, so I won't repeat it. But seriously, get and configure this module. It's hard to believe it's still not a standard part of most distros.

GET

Everyone you watch working on a site starts with the GET method on the address bar – but almost everything is written to use the POST method. I have inserted this section of the script:

```
my $c = new CGI;  
print $c->header();  
my $method = $c->request_method();  
die "Method $method not allowed" unless ($method =~ /POST/);
```

And you can see that if anyone tries to override the POSTed methods by passing an appropriate URL, the program dies. Make sure you place this just after the command that prints the header, so that the error catching code won't execute before the HTTP header is sent.

Appendix Code

DEMO USER:

```
[oracle@yourmum database]$ sqlplus '/ as sysdba'  
SQL> CREATE USER WEBSQL  
 2 IDENTIFIED BY SQLPASS  
 3 QUOTA 0 on SYSTEM  
 4 QUOTA UNLIMITED ON USERS  
 5 DEFAULT TABLESPACE USERS;  
User created.  
SQL> GRANT CONNECT, RESOURCE to WEBSQL;
```

Note that because I had my caps lock key on, I just made the password all caps too.

DEMO TABLE

```
[oracle@yourmum database]$ sqlplus websql  
SQL*Plus: Release 11.1.0.6.0 - Production on Wed Jun 18 11:52:54 2008  
Enter password:  
Connected to:  
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production  
With the Partitioning, OLAP, Data Mining and Real Application Testing options  
SQL> CREATE SEQUENCE USERKEY  
 2 INCREMENT BY 1 START WITH 1;  
Sequence created.  
SQL> CREATE TABLE USERS (  
 2 id NUMBER PRIMARY KEY USING INDEX TABLESPACE indexes,  
 3 pass VARCHAR(64),  
 3 name VARCHAR(32) NOT NULL UNIQUE,  
 4 private_info varchar(128) );  
Table created.  
SQL> CREATE OR REPLACE TRIGGER USERKEYINSERT  
 2 BEFORE INSERT ON users  
 3 FOR EACH ROW  
 4 BEGIN  
 5 SELECT userkey.nextval INTO :new.id FROM dual;  
 6 END;  
 7 /  
Trigger created.
```

WEB SITE DBWORK.PL CODE:

```
#!/usr/bin/perl -wT  
use CGI qw(:standard);  
use CGI::Carp qw(fatalsToBrowser);  
use DBI;  
use DBD::Oracle;  
  
use strict;  
  
$/++;
```

```

my $c = new CGI;
print $c->header();
my $method = $c->request_method();
die "Method $method not allowed" unless ($method =~ /POST/);
my ($dbh, $sth, $sql);

my $pass = 'SQLPASS';

$dbh = DBI->connect('dbi:Oracle:host=localhost;sid=mumlol',
    'websql', $pass, {
        RaiseError => 1,
        AutoCommit => 0
    }
)

$sql = $c->param('sql') or die "No SQL presented";

eval {
    $sth = $dbh->prepare($sql);
    $sth->execute();
};
die "Failed SQL with: $DBI::errstr" if ($@);

print "Output for SQL Command $sql was<P>";

while(my @rows = $sth->fetchrow_array() )
{
    print "Row: @rows<P>"
}

$sth->finish();
$dbh->disconnect();
my $filename = $c->param('includefile');
if ($filename && -e $filename)
{
    open FH, $filename or die "Unable to open include file @!";

    print "File $filename contents are:\n";
    print $_ while (<FH>);
    close FH;
}

```