

University IT Security Office Standard: Web Applications

Version 1.0

Effective: September 1, 2009

1. Definitions

Web application or web-based application: An application that is accessed via a web browser over a network, or is dependent on a browser to execute code.

2. Purpose

The purpose of this standard is to assist Duke developers and administrators to implement secure web-based applications. Compliance with this standard does not exempt a web application from meeting University, federal, or state regulations or other required standards. (For example, if a web application is accepting credit card data, then the application and server must comply with all PCI policies implemented by Duke's Treasury and Cash Management Office.)

As there are a wide variety of languages, tools, and system configurations used across campus, this document is not intended to be a "How-To" on application security. Instead, this standard is intended to note issues to be addressed and documented in a manner which is appropriate to the services provided. It is expected that all applications will comply with the requirements of this standard before being put into production. All recommended practices listed should be implemented unless they significantly disrupt business operations. If the standard is customized for a unique business operation, all customizations should be documented.

Effective implementation of this standard should minimize the likelihood of unauthorized access to University computing resources and protected data. However, all security events must be reported to security@duke.edu as soon as they are discovered, in order to ensure compliance with legal obligations.

3. Scope

This standard applies to all web-based applications (as defined above) administered, developed, or serviced by Duke staff or by third parties via contractual agreements. It applies to anyone who owns, administers, or maintains an official Duke web application that hosts or provides access to Duke-classified Sensitive or Restricted information. The standard also applies to all Duke web applications that use Duke authentication services, as Duke credentials must be properly protected.

4. Required

4.1 Protect Confidential Data

University web applications that process Sensitive or Restricted information must follow the control requirements of the Duke University Information Classification Framework. Sensitive and Restricted data (as defined in the framework) are prohibited from being stored persistently on Internet accessible (i.e., public facing) web servers. Please refer to that document (posted on security.duke.edu) for the most current definitions.

Examples of data that would typically be classified as Sensitive include:

- Protected Health Information
- Student Educational Information
- Financial Information
- Driver's license numbers or Social Security numbers

Examples of data that would typically be classified as Restricted include:

- Library transactions
- Transactions covered by non-disclosure agreements

4.2 Protect Against Common Web Application Exploits

Web applications should be developed to prevent common exploits. Common exploits and programming errors are listed by several reputable organizations, including OWASP and SANS (see Resources section below).

4.3 Checklist for Securing Web Applications

Developers of publicly accessible web applications (whether developed by Duke personnel or by a third party) should use the accompanying checklist (in Appendix B) to assist them in securing those applications. The checklist includes industry standard security settings for web applications and requires a statement of business need for those applications that are public facing. The checklist also includes security settings for backend database connectivity used by the web application. The checklist should be completed by the business owner who is responsible for the application. The signed checklist should then be included in audit documentation and updated as necessary. Externally developed web applications that reside on Duke web servers should abide by the same process and be subject to the same requirements as applications developed by Duke staff or faculty.

5. Recommended

5.1 The following practices are recommended for all web **applications**:

- Include security in the initial design phase of the software development life cycle (SDLC).
- Use a sandboxed or managed code environment for development.
- Use de-identified data for development and testing.
- Disable path traversal and directory browsing; place index files in each of the web directories. You can also look closer at your specific server settings such as setting permissions in an .htaccess file.
- Encrypt any data passed to a client. (Data passed to a client, such as cookies, session ids, or hidden fields are modifiable by the client.)
- Follow the principle of least privilege, meaning that applications have access to do only what they need to.
- If the application needs a username and/or password to log into some other service as part of its function, those credentials should not be stored in the same file as the program source code.
- If the application stores user password information, it must do so as a salted, cryptographically secure hash.
- Applications should not execute OS shell commands, and especially not pass user input to the shell.
- Use a standard input validation mechanism to validate all input data for length, type, syntax, and business rules before accepting the data to be displayed or stored. Use an "accept known good" validation strategy. Reject invalid input rather than attempting to sanitize potentially hostile data. Do not forget that error messages might also include invalid data
- If a web application is using Duke credentials, verify that all exchanges of usernames/passwords are encrypted from the web server to other Duke servers.
- A code review by someone other than the developer(s) should be performed before an application is moved into production.
- Remove comments and commented code from all production applications.
- Audit web directories for unused files and remove them.
- Remove sample scripts from all production applications.
- Remove development tools from all production applications.
- Log all access to the application, including: authentication and access control, account lockouts, accessed resources and reasons for denial of access, session termination, policy violations, data reads, writes, and deletes, failed queries, file not found errors, unexpected states, db connection failures, timeouts or performance related messages, referrer entries. Logs entries should include user id, timestamp, source ip, description of the event, and error codes. When setting up logging, think about

correlation with other logs in the case of a security incident, and ensure that the system time is synchronized with an authoritative source. If using a logging API, use a commercial or open source API rather than writing your own.

- Maintain and monitor all web access logs; follow your internal policy on log creation, retention, rotation, and audit.
- Avoid generic accounts (such as DukeUser1) in production applications.
- Restrict or remove administrative access to production applications when possible. If administrator's access is built-in (for applications that require such access) consider stricter controls for such access.
- Do not rely on robots.txt files for security – it is a map to interesting pages for some users.
- Do not post production code that might provide configuration information to public mailing lists.
- Protect against all the top vulnerabilities on the current OWASP list.
- Web application programmers/developers must keep up-to-date with emerging security threats that affect web applications and should complete annual security awareness training on secure web application development. It is recommended that web application developers achieve (and keep current) certification for secure programming or web application security through SANS or other professional organizations.

5.2 The following practices are recommended for all web **servers**:

- Isolate and harden web and database servers according to the CIS benchmarks and the ITSO server standard. Additional layers of security can be added, depending on the OS and application framework (mod_security or urlscan, for examples).
- Include the web application framework in the patch cycle (php, ruby, etc).
- Use a web application firewall.
- Deny HTTP PUT; allow only GET and POST.
- Deny access for the web server account to other parts of the web server file system.

6. Enforcement

It is the responsibility of application owners to ensure that the controls described in this document are implemented. It is expected that the secure implementation of web-based applications is a part of everyone's job within the department.

Departmental applications will undergo periodic internal and external audits. Internal audits are carried out by the Office of Internal Audits. The initiation of an internal audit should be based on a risk analysis, also performed by the Office of Internal Audits. A requirement for an external audit may be recommended as a result of the internal audit, or be requested independently by a department's management. The department is responsible for remediation of any findings of non-compliance with this standard within the time frame indicated by the auditors.

Based on industry standards and practices:

SANS Software Security Institute

Center for Internet Security benchmarks (cisecurity.org)

Open Web Application Security Project (OWASP)

National Institute of Standards Technology (NIST)

Review Frequency: Annually

Updated: 7/09

Authority:

Duke University Chief Information Officer
Duke University Chief Information Security Officer

In Compliance with:

Duke University Information Classification Framework
ITSO Information Security Standard: Server Security

Other resources:

University IT Security Office website: <http://www.security.duke.edu>

Center for Internet Security: <http://cisecurity.org>

Open Web Application Security Project: <http://www.owasp.org> (Top 10 Exploits)

SANS Institute: <http://www.sans.org> (Top 25 Programming Errors, training classes)

SANS Software Security Institute: <http://www.sans-ssi.org>

If you are using a Service Oriented Architecture framework, the NIST SP800-95 (Guide to Secure Web Services) can be very helpful.

For help with reviewing code, the OWASP ebook “OWASP Code Review” is helpful.

Appendix A Code Examples

1. SQL Injection

SQL injection occurs when a user can manipulate data that is sent to your database and thus gain control over the database. A user can do this through any of the following routes and thus all values from these locations must be checked:

- Cookies
- Sessions (stored in cookies, generally more useful in other hacking techniques)
- GET or POST variables (Request.QueryString[] and Request.Form[] in .NET)
 - Note this is not limited to values in which you ask the user to input. All values that are passed through this method can be changed by the user, including hidden values.
- Environment variables such as the referrer or user agent
- HTTP headers

The most general way to avoid SQL injection is to write stored procedures rather than passing SQL to the database, as this method inherently protects you. However, if you must pass SQL, please follow the language specific recommendations below.

Please note it is especially important to hide raw database errors from users because these usually give information that would be very useful for gaining access to the database.

1.1 PHP

Whenever taking input, it is important to check the input as specifically as possible. There are a few ways of doing this. In the case of simply expecting an integer use the `is_numeric()` function, which returns a boolean. However, all other inputs to SQL, if using the MySQL database, should be passed through `Mysql_real_escape_string()` function as demonstrated below.

```
<?php
    $dbhost = "hostname" ; //most likely localhost
    $dbUser = "username" ;
    $dbPW = "password" ;
    $database = "databasename " ;
    $dbh = mysql_connect ( $dbhost , $dbUser , $dbPW)
        or die ( " I cannot connect to the database because : " .mysql_error ( ) ) ;
//Connection to the database is necessary to run mysql_real_escape_string
mysql_select_db ( $database ) ;
$username = mysql_real_escape_string ( $GET [ ' username ' ] ) ;
$query = "SELECT * FROM Users WHERE username = ' " . $username . " ' ; " ;
$res = mysql_query( $query ) ;
?>
```

When using PostgreSQL, a similar approach is used. Again, a connection to the database is necessary and then use either `pg_escape_string()` or `pg_escape_bytea()` depending on the data type.

1.2 Perl

In Perl, the most common method of interacting with databases is through the DBI class. Within this class there is a simple parameterization method using the `prepare()` and `execute()` methods in which all inputs are sanitized. A basic example of this is shown below.

```
my $dbh = DBI->connect ( 'DBI : mysql :mydata ' , 'me ' , 'mypass ' ) ;
```

```

my $queryHandle = $dbh prepare (q{
SELECT username , password , dateRegistered
FROM Users
WHERE posts > ?
email = ?
} ) ;
$queryHandle >execute (10 , ' fakeEmail@duke.edu ' ) ;

```

In the process above, a question mark is used to indicate all query parameters, and then all parameters are passed through the execute command where they are sanitized before being passed to the database.

1.3 Python

In Python there is no standard module for all of the different database systems and as a result no definitive method to always use for preventing SQL injection. Using the common MySQLdb module, an example of parameterization is shown below:

```

#!/usr/bin/env python
# encoding : utf-8

import MySQLdb

def main ()
    db = MySQLdb.connect ( host="localhost " , user="username " , passwd="password" ,
    db="databaseName" )
    cursor = db.cursor ()
    cursor.execute( "SELECT _ FROM Users WHERE email = '%s ' AND pw = '%s ' " ,
    ( email , password ) )
    result = cursor.fetchall()
If __name__ == '__main__':
    main ()

```

The exact function may vary depending on the package you are using, but the concept and workings of the parameterization are the same, and will protect you from SQL injection.

1.4 .NET

MS SQL with SQL Studio Manager has easy to use Stored Procedures that should be used whenever possible, as passing values this way automatically sanitizes them from SQL injection. If you must send a SQL query directly to the database, use the String.Replace() method to get rid of or appropriately escape values such as quotations, commenting symbols like /* and --, new lines, etc. In general, validate your input as specifically as you can and use Stored Procedures whenever possible.

2. Cross Site Scripting (XSS)

As with SQL injection, all input (cookies, GET or POST variables) can be used for cross site scripting depending on how they are displayed to the user. Therefore, it is important to sanitize all input for XSS.

2.1 PHP

To avoid XSS all input that will be displayed to the user should use:

```

$output = htmlentities($input);

```

```
//or....
$output = htmlentities($input, ENT_QUOTES, "UTF-8");
```

2.2 Perl

When using Perl, it is important to parse out all HTML Entities to protect against XSS. Some examples of this can be seen below.

```
#!/usr/bin/perl
use HTML::Entities;
print "Here you go user " .HTML::Entities::encode($name).", enjoy!";
```

Another basic example for mod perl is:

```
use Apache::Util ;
use Apache::Request ;

my $apr = Apache::Request->new(Apache->request);

my $text = $apr->param('text');

$r->content_type("text/html");
$r->send_http_header;
$r->print("You entered ", Apache::Util::html_encode($text));
```

2.3 Python

To escape the <, >, &, and quotation mark characters, use the quoteattr function from xml.sax.saxutils:

```
import cgi
from xml.sax.saxutils import quoteattr

print "<a href='\" + quoteattr("#' onmouseover='alert(\\'XSS\\')' ") + "\">Link</a>";
```

2.4 .NET

The .NET framework is able to do much of the heavy lifting as far as base level sanitizing user input. Because of post backs there is less of a need for hidden variables. Also make sure to have in your web.config file:

```
<system.web>
<pages validateRequest="true" />
</system.web>
```

To sanitize individual inputs, you can also use `HttpUtility.HtmlEncode()`.

3. Hashing

It is important to not store plain text of certain sensitive information. In these cases, make sure to use different hashing schemes. Some are listed below. In general, each language has an implementation of the md5 hash. However, in the modern age of Google, many common md5 hashes can be decrypted by simply searching for the hash value on Google. For example, search f9f16d97c90d8c6f2cab37bb6d1f1992 and you will see many results for doctor, thus defeating the purpose of a hash. To protect against this, it is important to use a salt (a string that will be always used for hashing to make your hash unique). This can be done in one of two ways. For one, append your salt to the beginning of all strings before implementing the md5 hash. This will make your hash different and thus defeat the ability to Google the value and decrypt the hash. The other option is that some languages have specific functions which, when passed a salt, will use that salt in the hashing process. Both of these options are shown below.

3.1 PHP

For a basic md5 hash, use `md5()`. To have a more unique encryption use:

```
$hash = crypt($input, $salt);
```

In this case, create a unique salt value that will always be used for your application and will create a unique hash. The salt should be a string ranging from 2 to 12 letters long.

3.2 Perl

To do a basic md5 hash, follow the methodology below:

```
#!/usr/bin/perl
use Digest : :MD5 qw(md5 md5_hex md5_base64 ) ;
$data = " doctor " ;
$hash = md5_hex $data ;
print $hash . "\n" ;
```

Another method, which takes a salt as a parameter, is the `Crypt::UnixCrypt XS` as shown below:

```
use Crypt : : UnixCrypt_XS qw/crypt / ;
$hash = crypt ( "MYPASSWORD" , "SALT" ) ;
```

3.3 Python

An example of a basic md5 hash can be seen below.

```
import hashlib

test = hashlib .md5( " doctor " ) ;
print test.hexdigest ( )
```

The `hashlib` library used above contains a number of other hashing algorithms such as `sha1`. To use a salt in your encryption, follow the example below.

```
import crypt

print crypt.crypt ( "doctor " , " salt " )
```

3.4 .NET

To do an md5 hash in .NET requires use of the `MD5CryptoServiceProvider` class. A C#.NET implementation of this class can be seen below from <http://blog.brezovsky.net/en-text-2.html>

```
Public string GetMD5Hash(string input) {
    System.Security.Cryptography.MD5CryptoServiceProvider x =
        new System.Security.Cryptography.MD5CryptoServiceProvider ( ) ;
    byte[] bs = System.Text.Encoding.UTF8.GetBytes(input);
    bs = x.ComputeHash(bs);
    System.Text.StringBuilder s = new System.Text.StringBuilder();
    foreach(byte b in bs) {
        s.Append(b.ToString("x2").ToLower());
    }
    string password = s.ToString();
}
```

```
    return password;
}
```

A similar methodology can be used in VB.NET. Unfortunately, there is no crypt class naturally in .NET as in the other languages outlined above. Therefore, to use a salt you can either append the salt to the original key before performing the md5 hash as described above, or there are some independently created crypt classes that can be found by searching online. Regardless, it is important to keep in mind the uniqueness of your hash depending on the type of data you are hashing.

4. Cross Site Request Forgery (CSRF)

CSRF hacks are fairly different than the previously mentioned hacks and much harder to completely prevent. A CSRF hack is executed by a request (either POST or GET) being executed from an outside site. While many valid and useful web design practices, such as RESTful API's, utilize GET and POST functions, these requests can also be used to exploit users. This is generally done by a third party site executing a GET or POST as a user that is currently logged in and authenticated on your site. This may be done within an image tag such as:

```
<img src = "http://yourbankingsite.com/transfer.php?givemoneyto=0122534" />
```

Here the image tag will of course not show an image but will still send the request to your site and if the user is logged in may be executed unless properly guarded against. Javascript on an external page can accomplish a similar effect.

Another method for the attack that bypasses some of the solutions below is to include the page as an invisible iframe. To prevent this include the following Javascript:

```
<script type="text/javascript">if (top != self) {top.location.href = self.location.href;}</script>
```

However, this will only work if loading the page does not perform the action, since server side code will be executed and loaded before the Javascript is run.

For each of the languages below, there are examples of how to check the request type, the referring address, and set up unique tokens for each user to confirm that requests are coming from the actual user. An explanation of why each of these defenses is important can be seen in the PHP section below.

4.1 PHP

To avoid CSRF attacks a first line of defense is to use POST rather than GET for any input that will cause a change. When receiving this input on the server make sure to use `$_POST['key']` rather than, `$_REQUEST['key']`, which will accept both POST or GET inputs. However, Javascript on an external page can execute both POST and GET requests. To prevent this, the server can also check the address from which the request was submitted and make sure it was from your own website. This value can be gotten from `$_SERVER['HTTP_REFERER']`. While this may prevent some CSRF attacks, Javascript can fake the referrer too, and thus for certain interactions may not be enough protection. The only way to truly prevent this is to have a random key saved on the server (either in the session or in a database) for each user and included with every request. This key would have to be checked before each action and expire after a short amount of time. An example of this can be seen below:

```
$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;
$_SESSION['token_time'] = time();
```

Then include this token in form submits with:

```
<input type='hidden' name='token' id='token' value='<?php echo $token; ?>' />
```

With this, each form submit should check this token against that stored in the server session:

```
<?php
if($_POST['token'] == $_SESSION['token']) {
    /* Perform actions */
} else {
    /* CSRF attempt or timed out */
}
?>
```

And the token should be reassigned after a given period of time:

```
<?php
$token_age = time() - $_SESSION['token_time'];
if ($token_age <= 300){
    /* Less than five minutes has passed. And key is still good*/
    if($token_age >= 200) {
        /* if this long ago, reset the token and the time. */
        $token = md5(uniqid(rand(), TRUE));
        $_SESSION['token'] = $token;
        $_SESSION['token_time'] = time();
    }
}
?>
```

With this system it is much harder to perform a CSRF attack.

4.2 Perl

As mentioned above, a first line of defense is to make sure you are only receiving POST request variables and not using GET variables. Therefore make sure to check the request type with `$ENV{'REQUEST_METHOD'}`. To check the referrer for a request you can similarly use `$ENV{'HTTP_REFERER'}`. Finally, to use the token methodology described above, you can use pieces of code from below.

```
#!/usr/bin/perl

use CGI::Session;
use CGI;
use Digest::MD5 qw(md5 md5_hex md5_base64 );

$cgi = new CGI;

#this keeps sessions consistent
$sid = $cgi->cookie('CGISESSID') || $cgi->param('CGISESSID') || undef;
$session = new CGI::Session(undef, $sid, {Directory=>'/tmp'});

#for the basic assigning of token variables
$token = int(rand(10000)) * time;
$token = md5_hex $token;
$token_time = time;

$session->param("token", $token);
```

```

$session->param("token_time", $token_time);

#to later access session variables
$token = $session->param("token");
$token_time = $session->param("token_time");

#and then to check the age of the token
$token_age = time - $token_time;
if ($token_age <= 300){
    #Less than five minutes has passed. And key is still good
    if($token_age >= 200) {
        #if this long ago, reset the token and the time.
        $token = int(rand(10000)) * time;
        $token_time = time;
        $session->param("token", $token);
        $session->param("token_time", $token_time);
    }
}

```

With these basics it is much harder for a CSRF attack to be executed.

4.3 Python

Using the CGI module, there seems to be no good method of checking the type of form requests, and as a result there should be a higher weight on the other methods of protection. To check the referrer for a request you can use `os.environ['HTTP_REFERER']`. To do the token method used above is very hard without a python framework because it requires the use of sessions, which are not native to python. However, many frameworks offer strong protection against CSRF attacks. Django in particular has built in protection against CSRF attacks as outlined here: <http://docs.djangoproject.com/en/dev/ref/contrib/csrf/>. It is strongly recommend that you build your applications on top of one of these frameworks to make CSRF protection easy and your application secure.

4.4 .NET

In .NET the simplest way to completely avoid CSRF attacks is using the enabling ViewState with including the following tag in the Web.config:

```
<pages enableViewState="true" />
```

However, because this can be disabled in individual pages, it is worthwhile looking at other methods for deterring CSRF attacks. As mentioned in all of the examples above, it is important to make sure that for a POST request, to only accept POST variables and not GET variables. To do this use `HttpRequest.Form['foo']` and not `Request.Params['foo']`. Also, to check the referrer for a given request, use `Request.ServerVariables("HTTP_REFERER")`. An additional simple check is to include on every page in `Page_Init` the following code (from http://www.owasp.org/index.php/.Net_CSRF_Guard):

```
ViewStateUserKey = Session.SessionID;
```

These basic protections should sufficiently guard against CSRF attacks.

5. E-mail header injection

E-mail header injections are very similar in nature to SQL injection attacks. As in SQL injection where users can write SQL code in their input, users can also modify e-mail headers through their input if not carefully parsed. This will allow your server and web application to be used as a portal for spam, using your server and

e-mail address as the source but with a custom subject, message, and even attachments sent to arbitrary e-mail addresses. This type of impersonation is of course dangerous and should be protected against by carefully parsing all user input being used in e-mail being sent. Unfortunately, how to protect against this is dependent on the language and method you are using. In general, you should use strict only allowing specific characters (not looking for “bad” characters as you will probably forget some).

The general advice is to use a pre-made library for sending emails and to check the specific library’s background with email header injections. In most cases for typical emails with no extra headers specified), the application should be careful with the “To,” “From,” and “Subject” headers. The key to doing this of course is to carefully parse anyplace in these headers that can have user input. To parse e-mails, there is a fairly simple regular expression to make sure that the input is a valid e-mail address and not dangerous:

```
^[+.\0-9=a-z_]+@[(-0-9a-z)+\.]+([0-9a-z]){2,4}$
```

Note this is not the RFC standard definition of an e-mail address but merely checks that the email address is not dangerous.

For subject it is much harder because there are no specific characters than can be allowed, but rather a specific few that shouldn’t. However, as stated above, checking which characters is not allow is much more dangerous. Therefore, it is recommended to not let users decide on the subject of the email. If it is necessary, the regular expression below should work most of the time:

```
\r\n
```

Below I will outline functions using these protections and other methods of protection specific to each language.

5.1 PHP

Using the common mail() function, the inputs \$to, \$subject, and \$additional_headers, which typically contains the From header, are vulnerable to email injection if not carefully checked. Below is an example of two functions to check user input before sending an e-mail with PHP.

```
<?php
```

```
//return true if header is safe
function simpleSafeHeader($input) {
    return (preg_match("/\r\n/", $input) == 0);
}

function validEmailAddress($input) {
    return (preg_match("/^[!#\%&'*-+.\0-9=a-z_]+@[(-0-9a-z)+\.]+([0-9a-z]){2,4}$/i", $input) == 1);
}

//We will assume all inputs are from the user and thus are dangerous
$to = $_GET['to'];
$from = $_GET['from'];
$subj = $_GET['subj'];
$message = $_GET['message'];
```

```

if(validEmailAddress($to) && validEmailAddress($from) && simpleSafeHeader($subj)) {
    mail($to, $subj, $message, "From: ".$from);
} else {
    echo "There are inappropriate characters in your input.";
}
?>

```

Zend and PEAR also offer alternative mail functions that are inherently safer and will do this for you.

5.2 Perl

If you are using the `/usr/sbin/sendmail` or some other variation of `sendmail` to email through Perl it is necessary to check the To, From, Subject, and other primary header fields. Below is a custom example with regular expressions to check email strings for being valid email addresses.

```

#!/usr/local/bin/perl

sub checkEmailAddress {
    my $a;
    if($_[0] =~ m/^[!#$%&'*-+\.0-9=a-z_]+@[(-0-9a-z)+\.]+([0-9a-z]){2,4}$/i) {
        $a = 1;
    } else {
        $a = 0;
    }
}

sub simpleSafeHeader {
    my $a;
    if($_[0] =~ m/r|\n/i) {
        $a = 0;
    } else {
        $a = 1;
    }
}

my $email = "local\@domain.com";
my $subject = "test subject";
if ( checkEmailAddress($email) == 1 && simpleSafeHeader($subject) == 1) {
    print "safe\n";
} else {
    print "not safe\n";
}

```

An alternative to the `checkEmailAddress` function above is CPAN's `Email::Address`, which is demonstrated below:

```

unless (Email::Valid->address($email) {
    print "You supplied an invalid email address.";
}

```

```
exit;
}
```

5.3 Python

If you are using a framework such as Django, then `django.core.mail` with the `django.core.mail.send_mail` function, then this issue is taken care of for you. However, if you are creating your own mail function use an appropriate regular expression function as shown below.

```
#!/usr/bin/env python
# encoding: utf-8

import re

# return true if valid
def checkEmailAddress(inValue):
    return (re.search(r"^[!#$%&*+/,=/?@^\\_`{|}~-+\\.0-9=a-z_]+@[(-0-9a-z]+\\.)+([0-9a-z]){2,4}$", inValue)
    == None)

def simpleSafeHeader(inValue):
    return (re.search("\r|\n", inValue) == None)

email = 'local@domain.com'
subject = 'some subject'
if(checkEmailAddress(email) & simpleSafeHeader(subject)):
    print "good"
else:
    print "no good"
```

5.4 .NET

Using `System.Net.Mail` with the `MailAddress` object for adding senders and recipients will take care of these headers and make sure there is no injection. However, if you are using other methods of sending mail, as there are multiple different ways in .NET, make sure to check if it is necessary to parse the input yourself. It can never hurt to restrict input with a regular expression check as explained above.

Appendix B Checklist

Web Application Security Checklist

Complete, sign, and date this checklist for each web application and update annually or as needed. Keep the completed form with the other application documentation.

Authentication

Authentication must be consistent for every part of the application so that there are no bypass possibilities. Centralized authentication should be used for access control.

Which centralized authentication service is used? _____

Is SSL in use? Yes No

Session Management

The application uses randomly generated long session IDs.

The application's sessions include an expiration time.

Length of session? _____

The application's sessions are destroyed on logout.

Logouts or timeouts are enforced.

Only session IDs are sent to clients (to minimize the amount of session information available to users).

Session tokens (whether cookies, URL based, hidden fields, or HTTP headers) are provided after authentication takes place, and are protected.

URL based session tokens are not used.

The only information stored on the client (in a cookie or hidden form field, for example) is the session ID.

(Most programming languages have built-in session management tools which should be used when possible, but not relied on exclusively for secure session management.)

Input Validation

Cross-site scripting vulnerabilities allow malicious attackers to take advantage of web server scripts that do not adequately filter data sent along with page requests to inject JavaScript or HTML code that is executed on the client-side browser. These flaws occur anywhere a web application uses input from a user in the output it generates without validating it. Any type of variable that comes from a user or comes from a place where you do not control needs to be validated. This malicious code will appear to come from your web application when it runs in the browser of an unsuspecting user.

Do not rely on client-side scripting to validate input (users can circumvent this easily); do additional server side checking.

Every instance of user input is fully validated.

If the application displays user supplied input, the data is displayed by a function that either escapes or converts the data into appropriate output.

Character set encoding is explicitly set for each page generated by the web server.

Special characters are identified.

Dynamic output elements are encoded.

Specific characters in dynamic elements are filtered.

Cookies are examined and validated.

All input is validated after Unicode, URL encode, and HTML encode has occurred, before processing.

A centralized validation function is used to validate user input (rather than having multiple validation functions).

Web applications that do not properly sanitize user input before passing it to a database system are vulnerable to SQL injection. This could potentially allow a malicious user to read and/or modify any data that the application has access to.

- The applications' scripts properly validate user input before passing it to the underlying database system.
- External interpreters are not used.
- Bind variables are used wherever possible. Where not possible, all user variables are escaped.
- Pattern matching is used to verify user input is an expected value.
- Access is limited for the web account that is accessing the database.
- Stored procedures are used to insert records and update data; the application does not have direct access to the tables.
- The application is limited to READ-only access where possible.

Web Application Database Configuration Requirements

Does the web application interface with any database servers? Yes No
If you answered 'No', skip the rest of this section.

If so, which database servers?

Is the database server housed on a physical server or VM on which it is the only function? Yes No
If not, what other functions are housed on the same server or VM?

Does the database server reside on a protected network? Yes No

The following requirements apply to database servers that interface with web applications:

- The Center for Internet Security (CIS) Security configuration settings for enterprise database servers should be used as a check of running systems and as a standard when configuring security settings for new systems.
- Web applications that connect to backend database servers must do so with encrypted connections so that the data and credentials are protected.
- Desktop database applications (eg. MS Access or FileMaker Pro) may not be used for web applications.

Security Logging Requirements

Is the web server centrally logging all security events? Yes No

How often are the logs reviewed?

Who reviews the logs?

Additional Requirements for Surveys Collecting Personally-Identifiable Data

Are Web survey subjects prevented from viewing any survey data other than their own records? Yes No
If yes, how is this accomplished?

Are survey respondents informed about any personal data collected? Yes No
If yes, how is this accomplished?

Do respondents have the ability to opt out of personal data collection or of not completing the survey before personal data are collected? Yes No

Does the survey site provide a privacy statement that describes the kind of information that is collected, how it is to be used, and how it may be disclosed? Yes No

Data Protection

List of any Sensitive or Restricted data processed by the application:

Authorized signature: _____

Date completed: _____

Updated: 08/09