Security in Modern Web Applications

Joshua Beam
Auburn University

josh@joshbeam.com

ABSTRACT
Of all types of software applications, web applications may be the ones for which security is of the most importance. Many web-based applications maintain databases of users and their personal information, allow the exchange of information among users, and may serve as official sources of information for individuals and organizations alike. Security breaches of a web application may lead to attackers harvesting large amounts of user information, eavesdropping on communications, and posting fraudulent messages that purport to be from users or system administrators whose accounts have been compromised.

Complicating the issue of web application security is the fact that a number of different potential targets exist for a single web application; in addition to the web application itself, the operating system, web server software, programming framework, database management system, and even unsuspecting users are potential weaknesses for a web-based service. A number of techniques, technical as well as non-technical in nature, have been developed to cut down on potential security vulnerabilities in web applications. While some attack vectors can be minimized through the use of modern web development frameworks and programming environments, web application developers must possess a knowledge of a number of security related topics in order to properly build and maintain secure web applications.

This paper will examine various security-related best practices employed by modern web applications. Topics covered will include setting up secure systems to host web applications on, techniques for storing and verifying passwords, minimizing the risk of social engineering attacks, and preventing technical attacks such as those involving cross-site scripting and SQL injection.

Categories and Subject Descriptors
K.6.5 [**Management of Computing and Information Systems**]:Security and Protection – a*uthentication, physical security, unauthorized access.*

General Terms
Security

Keywords
Web Applications, Operating Systems, Servers, Programming Languages, Frameworks

1.INTRODUCTION
Web application security can be fairly complex in that the security of such applications is dependent on not just the application itself, but of a number of other pieces of software that support the application. This paper will examine web application security at several levels, focusing on the technical implementation and deployment of modern web applications.

The first level is the operating system and the services it provides. A secure web application should be built for and deployed on an operating system that is up-to-date and configured to promote low-level system security. Additionally, there are some key services commonly used and packaged with the operating system that should be configured to mitigate security risks.

The second level consists of server software, and specifically two types of server software commonly used by web applications: the web server, also commonly referred to as the Hypertext Transfer Protocol (HTTP) server, and the database management system (DBMS) server. The former is the most public gateway to the system underlying a web-based application, and thus it should be set up with care in order to prevent unauthorized access to system resources, while the latter may contain large amounts of sensitive information, and thus its security is absolutely critical.

The next level consists of the application itself, along with any programming languages and frameworks used. While applications created with any programming environment are susceptible to attack, certain languages and/or frameworks may be intended to prevent certain categories of attacks or take steps to perform certain actions in a secure manner by default so that the application developer does not need to be as concerned with them.

On top of these three levels of the technical implementation of a secure web application, the human elements of security must be taken into account. The application should present a user interface that promotes the understanding by users of security policies that can mitigate the risk of social engineering attacks, and everyone involved in the operation of a web application should be aware of social engineering tactics in order to avoid being susceptible to them.

2.OPERATING SYSTEM SECURITY

The first level of security in the technical implementation of a web application is the operating system. The operating system ultimately provides the foundation upon which the other parts of a web application stand, and a misconfigured operating system could give an attacker access to every part of an application.

An overview of setting up a secure operating system configuration will be provided using Ubuntu, a Linux-based operating system that is popular for hosting web-based applications. Specifically, Ubuntu 13.10, the latest version available at the time of this writing, will be used.

2.1.Secure Shell (SSH) Configuration

The tool most commonly used to remotely access and administer servers is Secure Shell (SSH), and thus its proper configuration should be the first matter of business after the installation of the operating system.

One part of the SSH configuration that a system administrator should strongly consider changing for security reasons is the port that the service listens on. The default SSH port is 22, and thus attackers looking for systems to try to break into frequently scan for the availability of this port. Running SSH on port 22 is essentially advertising that the system is a potential target for attack [8].

The number of break-in attempts to a system can be significantly reduced by simply moving SSH to a different port [8]. Note that while this does not necessarily improve security (that is, it will do little, if anything, to deter an attacker determined to break into a particular system), it will decrease the visibility of the system from casual attackers.

The port can be set by modifying the "Port" setting in the /etc/ssh/sshd_config file on Ubuntu. Once the port has been changed, the sshd service should be restarted by running the following command:

```
sudo service sshd restart
```

A user attempting to log in to the system using the standard ssh command-line utility can then use the -p option to specify the proper port to connect to.

Note that when deciding on an alternate port to use for SSH, it is advisable to use a port below 1024, if possible. The reason for this is that ports below 1024 on Linux/UNIX can only be bound to by processes running as root [21]. If SSH were running on a port greater than or equal to 1024, there is a possibility that, in the event that the real SSH daemon stopped running for any reason, a malicious user with an account on the system could set up a fake daemon running on that port to obtain usernames and passwords [21].

The next aspect of SSH security to consider is what users require access to the system through SSH in the first place. In general, as few users as possible should be given SSH access. The SSH daemon can be configured to give access only to certain users by setting options such as "AllowUsers" in the /etc/ssh/sshd_config file [11].

The obvious example of a user that should almost never be given SSH access is root. This is the most powerful user account and its account name is no secret, so it's the prime target of attackers trying to break in to a system through SSH. Root logins can be disabled by setting the "PermitRootLogin" option in /etc/ssh/sshd_config to "no" [11].

It should be noted that, while Ubuntu sets this option to "yes" by default, logging in as root in general is not allowed in Ubuntu by default — users are required to use the "sudo" command to execute commands with super-user privileges. As a result of this, root SSH logins are effectively disabled by default.

2.2.Package and Update Management

Once a server's operating system has been installed and SSH has been set up properly for secure remote management, the administrator should take proper steps to ensure that the system has the software that it needs and that it remains up-to-date in the event that security updates are issued.

Ubuntu uses the APT package manager, which originated in the Debian GNU/Linux operating system that Ubuntu was originally based on. This package manager makes it easy to install and update software that has been packaged for Ubuntu. Installing the popular Apache web server, for example, is as simple as running the following command:

```
sudo apt-get install apache2
```

All packages installed through the package manager on Ubuntu can easily be updated by running two commands:

```
sudo apt-get update

sudo apt-get upgrade
```

The first command retrieves an updated list of packages available for Ubuntu, and the second command upgrades all installed packages to the latest versions. Ubuntu frequently updates packages to incorporate security fixes and other enhancements to applications.

The above two commands can be run periodically by the system's administrator to keep the system up-to-date. A system can also be configured to install updates automatically; Ubuntu includes a package to support this functionality, which can be installed by running the following command [5]:

```
sudo apt-get install unattended-
upgrades
```

Configuration files can then be modified to indicate what kinds of upgrades (such as security upgrades) should be installed automatically.

While Ubuntu's package manager is ideal for installing and updating certain types of software, such as the web server and database management system, it may be desirable to install some software through other means. One example would be Ruby and its associated libraries/frameworks. Ruby is the programming language that the popular Ruby on Rails web development framework is based on, and the version packaged with Ubuntu may not be as up-to-date as a developer would like, because the version packaged with Ubuntu is intended to be stable and simply support other software packaged with the operating system.

The Ruby community has developed various tools for installing and updating Ruby installations. One of the more popular solutions is "rbenv" [6]. This software can easily be installed by running a few commands, and it can be used to install any version of Ruby by running a simple command. It also allows multiple versions of Ruby to run side-by-side on the same system, which is useful for servers that must host multiple applications that were

developed using different versions of Ruby and/or Ruby on Rails, which may differ greatly from version to version [6].

3.WEB SERVER SECURITY

The next piece of software that should be installed and configured for hosting a web-based application is the web server. This section will use the popular Apache web server as an example.

3.1.Users and Groups

The first step in securing the web server software is ensuring that its users and groups are set up properly; in particular, the web server must be configured to run as its own, non-root user in order to minimize the potential damage that may result from the web server or any applications that it serves being exploited.

By default, Ubuntu runs the Apache web server using the "www-data" user and group, settings that can be modified by changing the APACHE_RUN_USER and APACHE_RUN_GROUP environment variables in the /etc/apache2/envvars configuration file; these environment variable are in turn used by the "User" and "Group" options in the /etc/apache2/apache2.conf file.

3.2.File System

A topic closely related to users/groups is file system security. While the "www-data" user has limited write access to the file system, it can still read many system configuration files that may be of interest to attackers. For this reason, Ubuntu's Apache configuration file ships with the following directives intended to secure access to the file system:

```
<Directory />
        Options FollowSymLinks
        AllowOverride None
        Require all denied
</Directory>
<Directory /usr/share>
        AllowOverride None
        Require all granted
</Directory>
<Directory /var/www/>
        Options Indexes FollowSymLinks
        AllowOverride None
        Require all granted
</Directory>
```

The first directory section above ensures that all paths beneath the root directory are, by default, not served by Apache, as indicated by the "Require all denied" directive within the section [1]. The second section uses the "Require all granted" directive to allow access to certain web-based applications packaged with the operating system, while the last section allows access to the /var/www directory that is the default location for user-installed web applications to be placed in [1].

The other directives contained in the above sections of the configuration file have security implications that are worth discussing. First, the "AllowOverride" directive is used to specify which Apache configuration directives can be overridden in .htaccess files contained within a subdirectory; using "AllowOverride None" is a secure default for the sections above, as it effectively disables the use of .htaccess files [1].

The other directive seen above is the "Options" directive, which is used to enable the "Indexes" and "FollowSymLinks" options for the /var/www directory. The "Indexes" option will cause Apache to serve directory listings when a user attempts to load a directory that does not contain a default document file (such as index.html) [1]. The effects of enabling this option can be seen by running the following commands:

```
sudo mkdir /var/www/test
sudo touch /var/www/test/hello.txt
```

After running the following commands, the URL http://localhost/test/ can be opened in a web browser on the server to see a directory listing including the hello.txt file. For security reasons, it may be desirable not to expose directory listings to users, and so the system administrator should consider disabling the "Indexes" option.

The "FollowSymLinks" option indicates that Apache should follow symbolic links that may point to another part of the file system [1]. The effects of this option can be seen by running the following command:

```
sudo ln -s /etc /var/www/etc
```

The URL http://localhost/etc/ can then be opened in a web browser on the server to see the contents of the system's /etc directory. Clearly this may be undesirable for security reasons, as a misconfigured symbolic link or a malicious user with access to a directory being served by Apache could allow users to access other parts of the file system through their web browsers. Removing the "FollowSymLinks" option from the "Options" directive and restarting the Apache server will cause an HTTP 403 Forbidden message to be served instead when trying to access the aforementioned URL through a web browser.

3.3.Secure Sockets Layer (SSL) Configuration

A common security measure implemented by web applications is to provide secure, encrypted access through Secure Sockets Layer (SSL). SSL is not enabled by default in Ubuntu's Apache configuration, but can be easily enabled by running the following commands:

```
cd /etc/apache2/mods-enabled
sudo ln -s ../mods-available/
socache_shmcb.load .
sudo ln -s ../mods-available/ssl.* .
cd /etc/apache2/sites-enabled
sudo ln -s ../sites-available/default-
ssl.conf 001-default-ssl.conf
sudo service apache2 restart
```

The first three commands set up the symbolic links required to have the Apache modules needed for SSL to be loaded when Apache starts, the next two commands enable the default SSL site configuration file included with Ubuntu's Apache installation, and the last command restarts the Apache server in order for the changes to take effect.

Using SSL requires the server to have access to both the public and private keys for an SSL certificate. Ubuntu includes a self-signed "snakeoil" certificate, which is adequate for testing or personal use. The following two lines in the /etc/apache2/sites-available/001-default-ssl.conf configuration file specify the paths to files containing the public and private keys, respectively:

```
SSLCertificateFile /etc/ssl/certs/ssl-cert-snakeoil.pem

SSLCertificateKeyFile /etc/ssl/private/ssl-cert-snakeoil.key
```

The public key, as its name implies, is to be publicly accessible and so it does not need to be secured in any way. The private key, however, must remain private; an attacker who gained access to the private key would be able to impersonate the party to whom the key belongs. For this reason, Ubuntu's "snakeoil" private key file is readable only by root, and any other certificates set up by the system administrator should have the same file permissions.

The system administrator may wish to create a new SSL certificate or use multiple certificates for different applications. An SSL certificate can be created using the OpenSSL command-line utility by running the following commands (based on examples in [14]):

```
sudo openssl req -new -x509 -nodes
-newkey rsa:2048 -days 365 -out /etc/
ssl/certs/ssl-cert-mycert.pem -keyout /
etc/ssl/private/ssl-cert-mycert.key

sudo chmod 600 /etc/ssl/private/ssl-cert-mycert.key
```

The first command generates a new SSL certificate that's valid for 365 days with a 2048-bit RSA key. The user will be prompted to enter information about their organization, contact information, and the fully-qualified domain name (FQDN) of the server that will use the certificate (see Figure 1). The FQDN entered should be the domain name that will be used to access the web application, since the certificate is associated with the FQDN to enhance security [14].

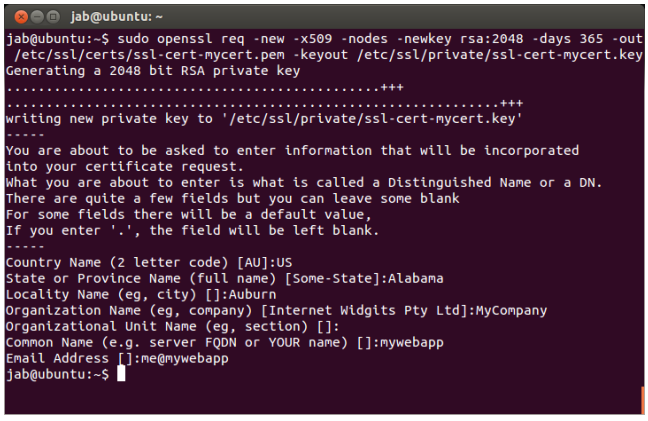

Figure 1. Generating an SSL certificate in Ubuntu.

The resulting certificate can be used as a self-signed certificate, but if the certificate is to be used for an application that other users will access, the system administrator will most likely want to have the certificate signed by an authority such as Verisign; in this case, the private key file generated can be used as a Certificate Signing Request (CSR) that can be sent to such an authority [15].

The SSL certificate itself specifies the public key cryptography cipher suite (such as the aforementioned 2048-bit RSA cipher suite), consisting of an encryption algorithm and its key size, that is used to create a secure channel to set up other aspects of an SSL session, including the symmetric key cryptography cipher suite that is used to perform the bulk of the secure communications. Apache can be configured to allow the use of a number of different cipher suites with the "SSLCipherSuites" directive [1]; Ubuntu uses the directive as follows in the /etc/apache2/mods-available/ssl.conf file:

```
SSLCipherSuite HIGH:MEDIUM:!aNULL:!MD5
```

"HIGH" and "MEDIUM" in the line above are collections of cipher suites that the OpenSSL library underlying Apache's SSL implementation considers to have high and medium levels of cryptographic security. The system administrator may wish to modify the above line if a specific level of security is desired. For example, websites that involve the transmission of very sensitive data may be required to use the AES encryption algorithm with a key size of at least 256 bits; the use of such a cipher suite could be forced with the following (using cipher suite names obtained from [12]):

```
SSLCipherSuite AES256-SHA:!aNULL:!MD5
```

The cipher suite setting can be verified by loading the site in a web browser such as Firefox; clicking the lock icon to the left of the address bar and clicking the "More Information" button will display a window that includes information about the cipher suite used for the connection (see Figure 2). The lower portion of the window shows information about the cipher suite used for the connection, including the encryption algorithm and key size.

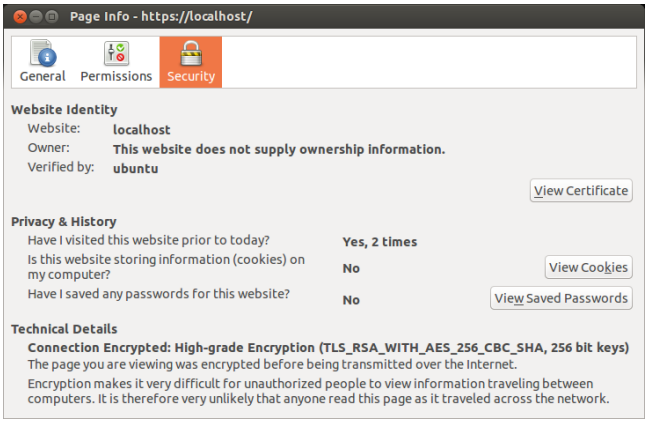

Figure 2. SSL connection information in Firefox.

While it may be desirable to use the most secure cipher suites available, it should also be noted that older browsers may not support such cipher suites, and so the system administrator may need to find a balance between security and compatibility.

4.DATABASE MANAGEMENT SYSTEM (DBMS) SECURITY

Securing the database management system (DBMS) is perhaps the key element of web application security. Even when implementing security measures for another part of the system, keeping the DBMS secure is often the ultimate goal; the database could contain large amounts of user information, and the potential for damage could be catastrophic if an attacker gained access to it.

4.1.Choice of DBMS

The first security choice related to the database is to decide what DBMS software to use in the first place. The choice of DBMS has certain security implications and the developer of a web application should consider what security features are desired before choosing a DBMS.

Consider SQLite, for example. SQLite is a popular DBMS choice for embedded systems and small applications in general due to its low overhead; SQLite is implemented as a library that is linked into applications and performs all functions related to accessing and manipulating a database [19]. This may not be optimal for security reasons, however; an application that requires access to a SQLite database must have file system access to the database's files and all data access/manipulation is performed on the system containing the web application itself, meaning that an attacker could theoretically gain full access to the database by exploiting some other weakness in the system to gain file system access.

A more secure choice for a web application would be a DBMS such as PostgreSQL or MySQL, each of which support the use of a client-server model. This would allow the database to reside on a machine separate from the web server and web application, providing an extra level of security. In the event that the system containing the web server and web application is compromised, the database may remain safe. In such a setup, the machine containing the database should be set up such that it is on a local network with the web server machine and cannot be accessed remotely by any machine over the internet, in order to prevent direct break-in attempts to the database.

By default, Ubuntu's MySQL installation allows connection attempts only from the local system by binding to 127.0.0.1, the IPv4 localhost address, as seen in the following option in the /etc/mysql/my.cnf file:

```
bind-address = 127.0.0.1
```

The above setting is a secure default for systems where the database server must remain on the same system as the application using the database; otherwise, this setting should be changed to allow connections from the web application.

The rest of this paper will discuss database security using MySQL 5.5 (the version included with Ubuntu 13.10), a popular open-source DBMS with native support in programming languages and frameworks such as PHP and Ruby on Rails.

4.2.Users and Passwords

MySQL can be installed on Ubuntu by running the following command:

```
sudo apt-get install mysql-server
```

The user will be prompted to provide a password for MySQL's built-in root user after running the above command.

Similarly to the web server, the DBMS server should be configured to run under its own, non-root system user account. Ubuntu runs the MySQL server as the "mysql" user by default, a setting that's configured with the following lines in the /etc/mysql/my.cnf file:

```
[mysqld]

user = mysql
```

The DBMS generally has its own set of users separate from operating system users, however, and this is generally the more important aspect of security when it comes to users in the context of a database. As mentioned above, MySQL has its own root user account; this user can perform any task involved in administration of the DBMS. An application using a MySQL database should have its own MySQL user account, which has only the permissions that the application must be able to perform.

MySQL includes a client application that can be used to perform database administration tasks. There are several ways to log into the MySQL server using this application; for instance, if the MySQL root user's password is "test", the following command could be used to log in:

```
mysql -u root -ptest
```

This is not considered a secure way of logging in, however; it's possible to see command-line arguments of running processes using the Linux/UNIX "ps" command, and commands that are run may be logged (the bash shell, for example, logs commands to the .bash_history file in the user's home directory), so the password should not be entered on the command-line [13]. A better way of logging in is to run the following command:

```
mysql -u root -p
```

With this command, the password is not provided on the command-line and the "mysql" program will instead prompt the user to enter the password for the root user [13].

As an example of creating a user account and giving it only the permissions it needs, we will create a database called MyWebApp, containing a "Users" table, by entering the following commands into the "mysql" client program:

```
CREATE DATABASE MyWebApp;

USE MyWebApp;

CREATE TABLE Users (Username
VARCHAR(100), Password VARCHAR(100));

INSERT INTO Users
VALUES('Administrator', 'test');
```

We'll now create a user called MyWebApp, with the password "test", to access this database:

```
CREATE USER MyWebApp@localhost
IDENTIFIED BY 'test';
```

Note that, for reasons similar to why you don't want to enter passwords on the command-line, MySQL commands may be logged and you may not want to enter the password in plaintext as above; an alternative is to provide the hash of the password instead when creating the user, by using the "PASSWORD" keyword [13]:

```
CREATE USER MyWebApp@localhost
IDENTIFIED BY PASSWORD
```

```
'*94BDCEBE19083CE2A1F959FD02F964C7AF4CF
C29';
```

The hash above is simply the hash of the string "test", which in this case was obtained by running the following command:

```
SELECT PASSWORD('test');
```

The following command can now be used to grant the user the ability to run SELECT commands to read information from the table:

```
GRANT SELECT ON MyWebApp.Users TO
MyWebApp@localhost;
```

The user can then run a query such as the following to access the table's data:

```
mysql> SELECT * FROM Users;
+---------------+----------+
| Username      | Password |
+---------------+----------+
| Administrator | test     |
+---------------+----------+
1 row in set (0.00 sec)
```

If the user were to attempt to insert data into the table, the following would occur:

```
mysql> INSERT INTO Users
VALUES('Admin2', 'test2');
ERROR 1142 (42000): INSERT command
denied to user 'MyWebApp'@'localhost'
for table 'Users'
```

The web application's developer should strive to give database users only the permissions they need in order for the application to function, and possibly use different DBMS user accounts for different parts of the application (for example, the part of the application that creates user accounts could run under a different user than the part that verifies the username and password during log in attempts).

4.3.Using Stored Procedures to Improve Security

In the example of the "Users" database table above, one obvious way of verifying a user's log in attempt might be to run a query such as the following, where the strings "@Username" and "@Password" represent the username and password, respectively, entered by the user:

```
SELECT COUNT(1) FROM Users WHERE
Username = @Username AND Password =
@Password;
```

If the count returned by the above query is not zero, then clearly a user with the given username and password exists, and therefore the log in attempt can continue.

The problem with this method, however, is that it may give the user access to more information than it requires; the user has SELECT permission for the entire table and thus is able to access all information in the table, which could be a problem if an attacker gained access to the user account.

A more secure solution can be achieved using stored procedures, which are small programs that execute on the DBMS and can be granted permissions independent of the tables that they operate on [7]. A stored procedure called VerifyLogIn could be created using the following commands:

```
delimiter ;;
CREATE PROCEDURE VerifyLogIn (IN
Username VARCHAR(100), IN Password
VARCHAR(100), OUT UserCount INT) BEGIN
SELECT COUNT(1) INTO UserCount FROM
Users WHERE Users.Username = Username
AND Users.Password = Password; END;;
delimiter ;
GRANT EXECUTE ON PROCEDURE VerifyLogIn
TO MyWebApp@localhost;
REVOKE SELECT ON TABLE Users FROM
MyWebApp@localhost;
```

The delimiter command is used to temporarily use two semicolons instead of one to indicate the end of a command entered into the "mysql" program (since a single semicolon is embedded in the stored procedure itself). After the stored procedure has been created, the MyWebApp user is granted permission to execute the procedure, and the user's SELECT permission on the Users table is revoked. The MyWebApp user can then verify a user's username and password using a command such as the following:

```
mysql> CALL
VerifyLogIn('Administrator', 'test',
@UserCount);
Query OK, 1 row affected (0.00 sec)
mysql> SELECT @UserCount;
+------------+
| @UserCount |
+------------+
|          1 |
+------------+
1 row in set (0.00 sec)
```

After calling the procedure, the @UserCount variable contains the number of users with the given username and password. The MyWebApp user can no longer access all information in the Users table, but can only verify whether or not a user with a given username and password exists.

4.4.Hashing Application User Passwords

The example of the Users table used above stores the hypothetical web application's user passwords in plaintext. This is poor practice from a security standpoint; an attacker who gains access to the table has a full list of all of the application's users and their passwords. Some of the passwords may have been used as passwords on other services used by the application's users, so such a breach has security implications for users outside of the application itself.

A better way of storing passwords is to store hashes of them, where the hashed forms are generated using a one-way hash algorithm such as SHA1. Because the algorithm is one-way, the

password itself cannot be easily derived from the hash (someone attempting to crack a password for which they have access to the hash would have to generate hashes for many possible passwords and compare the hashes to the hash of the desired password).

Our Users table can easily be updated to store SHA1 hashes of passwords instead of the passwords themselves by running the following command, using MySQL's built-in SHA1 function [13]:

```
UPDATE Users SET Password =
SHA1(Password);
```

The VerifyLogIn stored procedure could then be modified to use the SHA1 function on the password given as an argument to the procedure in order to compare the hash of the given password to the hashes stored in the Users table.

An even better method would be to generate a salt for each a password, a randomly-generated string of bytes that is appended to a password before generating its hash. The hash generated is then the hash of the password and its salt; the salt would be stored along with the hash, and the VerifyLogIn stored procedure would be modified to append the salt to the password given as an argument to the procedure before generating the hash for it. This makes the process of cracking the passwords stored more difficult for attackers by ruling out the possibility of using a dictionary of hashes for commonly used passwords.

4.5.Encrypting Data

It may be desirable to encrypt certain types of data; for example, if a user provides credit card information to an e-commerce application, the application must encrypt the credit card information in order to decrease the chances of an attacker gaining access to the information in the event that the database is compromised.

MySQL provides a number of functions to support the encryption and decryption of certain database fields; in particular, the AES_ENCRYPT and AES_DECRYPT functions perform encryption and decryption, respectively, using the AES algorithm with a key size of 128 bits [13]. Suppose that we wanted to add a credit card number field to the Users table and encrypt its contents; commands such as the following could be run:

```
ALTER TABLE Users ADD COLUMN
CreditCardNumber BINARY(128);

SET @Key = CAST(RPAD('MyKey', 128, '0')
AS BINARY(128));

UPDATE Users SET CreditCardNumber =
AES_ENCRYPT(CAST('1234567812345678' AS
BINARY(128)), @Key);
```

The following query could then be run to retrieve the decrypted contents of the CreditCardNumber column:

```
mysql> SELECT
AES_DECRYPT(CreditCardNumber, @Key)
FROM Users;
+-------------------------------------+

| AES_DECRYPT(CreditCardNumber, @Key) |

+-------------------------------------+

| 1234567812345678                    |

+-------------------------------------+
```

Of course, when using functions such as these, securing the keys used for encryption is of utmost importance. One option would be to store keys in a database table, preferably that is on a separate machine from the main database, that is properly secured and use a combination of stored procedures and user permissions to limit access to code that needs the keys. Unique keys should also be generated and used for each user, so that the compromise of one user's key does not affect other users' information.

For applications that require more security than that provided by the encryption functionality of the DBMS, data should be encrypted at the file system and/or application levels instead, for which a number of options are available.

5.APPLICATION SECURITY

The next level of security to be considered is application level security — that is, the security of the web application itself, rather than the security of the lower-level software that supports it.

5.1.Choice of Programming Language/ Framework

One of the first things to consider when evaluating the security requirements of a web application is the server-side programming language and/or framework to use to develop it. While proper security practices must always be followed by the developer regardless of the programming language or framework used, some languages are better suited for web development than others and some frameworks may provide protection against certain classes of attacks.

High level scripting or interpreted languages are typically used for programming web applications. While lower level languages such as C and C++ can be used, they are susceptible to buffer overflow attacks that higher level languages are usually not vulnerable to. Considering that web applications deal almost exclusively with the exchange and processing of user-provided data, and the result of a buffer overflow could be the compromise of a tremendous amount of user information, this is a major consideration and lower level programming languages such as C and C++ are generally considered unsuitable for web application development.

Popular programming languages for server-side web development include PHP, Java, C#, Python, and Ruby. Various frameworks have been developed on top of such languages; ASP.NET is a commonly-used C#-based framework, Django is built on Python, and Ruby on Rails is, as its name implies, built on Ruby. Some frameworks encourage programming practices or provide functionality to help mitigate the risks of certain types of attacks, such as code injection and request forgery attacks, which will be discussed later.

5.2.Hiding Debugging Information

Web programming frameworks often generate debugging information that is displayed in the web browser when an error occurs. This information is useful for developers when building the application, as it provides context as to what part of the code caused an error to occur.

This information can include sensitive information about the inner working of the application, however, and so it should be shielded from the view of outside users. The amount of information divulged in such debugging messages varies depending on the programming language and/or framework; PHP, for example, typically does not reveal much information (see Figure 3), while ASP.NET presents what's commonly known as the "yellow screen

of death," a verbose page exposing source code and other information [10].

PHP's configuration file includes the "display_errors" option which can be used to configure whether or not error messages should be displayed [16]. Ubuntu's PHP configuration disables the display of errors messages by default; developers may need to

Figure 3. PHP error page.

enable it during the development and deployment of applications, however, and should be sure to disable the option once an application has gone live. ASP.NET applications can easily be configured to display custom error messages to remote users while displaying debugging information only when running an application locally, an ideal solution for a developer working on an application that is already in use [10].

5.3.SQL Injection

One of the most common vulnerabilities found in database-driven web applications is the SQL injection vulnerability, a type of code injection vulnerability involving SQL database queries and commands. When executing a database query that incorporates some form of user-provided input, steps must be taken to make the input suitable for use within SQL queries, otherwise the user will be able to craft their input in such a way that they can execute arbitrary SQL statements of their choosing.

Consider a PHP script that must load a user's information from the database (using PHP's mysqli class) when the user attempts to log in. The simplest way to retrieve such information using the username and password entered by the user would be something like the following:

```
$query = $db->query("SELECT * FROM
Users WHERE Username = '" .
$_POST["username"] . "' AND Password =
'" . $_POST["password"] . "'");
```

The values of the "username" and "password" HTTP POST variables containing the username and password entered by the user are embedded within the SQL query that is executed. If the user entered the expected type of input for both fields, such as the string "Administrator" for the username and "test" for the password, the resulting query executed would be something such as the following:

```
SELECT * FROM Users WHERE Username =
'Administrator' AND Password = 'test'
```

If the query above returns any rows, then the application may proceed with logging the user in. The problem is that the user may instead enter malicious input into the username and/or password fields instead. For example, if the user entered the following password instead:

```
test' OR '' = '
```

The result would be that the following SQL statement would be executed:

```
SELECT * FROM Users WHERE Username =
'Administrator' AND Password = 'test'
OR '' = ''
```

In the example above, the user of the web application entered a password such that a WHERE clause that is always true has been injected into the query. The result is that the query returns one or more rows regardless of whether or not the username and password are correct. The result is that a user aware of the vulnerability can log in without knowing a correct username/password combination.

The simplest way to prevent such an attack is to use mysqli's escape_string function, which inserts the appropriate single-quote characters into a string containing user input to prevent attacks such as the above [16]. The PHP code above can be modified such that it looks like the following:

```
$query = $db->query("SELECT * FROM
Users WHERE Username = '" . $db-
>escape_string($_POST["username"]) . "'
AND Password = '" . $db-
>escape_string($_POST["password"]) .
"'");
```

As a result of this change, the same malicious user input described above would cause the following query to be executed:

```
SELECT * FROM Users WHERE Username =
'Admin' AND Password = 'test\' OR \'\'
= \''
```

The backslash characters inserted before each single-quote character entered by the user cause the single-quote characters entered by the user to be considered part of the password that is being searched for.

The above solution to the vulnerability works, but the developer must be careful to ensure that all user-provided input is escaped using the aforementioned function. This may be difficult to ensure for an application that works with a lot of user-provided input. A better solution would be to embed such input into queries in a different manner, one that frees the developer from having to manually call the escape function.

A better solution is to use prepared statements, in which placeholders representing parameter values are replaced with the values of the parameters themselves. The PDO class in PHP provides such functionality [16]. The above PHP code could be written instead as follows after connecting to the database with the PDO class instead of the mysqli class:

```
$statement = $db->prepare("SELECT *
FROM Users WHERE Username = :username
AND Password = :password");

$statement->bindParam(":username",
$_FORM["username"]);

$statement->bindParam(":password",
$_FORM["password"]);
```

```
$query = $statement->execute();
```

Prepared statements involve more code, but user input is embedded within SQL statements in a different way such that there's less of a chance of the developer directly concatenating input with SQL statements. The PDO class performs the proper escaping of input itself when the bindParam function is used, so as long as the developer is sure to use prepared statements when inserting parameter values into SQL statements, an SQL injection vulnerability is less likely to be introduced.

Another solution is to use an Object-Relational Mapping (ORM) framework, which maps database objects to programming language classes and objects in a way that allows developers to write database access code in their programming language of choice instead of embedding SQL statements within their code [18]. This is an increasingly popular solution, with Ruby on Rails including the ActiveRecord ORM framework, ASP.NET using the Entity Framework, and a number of ORM frameworks being developed for other languages such as PHP and Python. Eliminating manually written SQL statements eliminates vectors for SQL injection attacks; the downside in terms of security is that such frameworks may not be easy to use with fine-grained database object permissions such as those described earlier in the paper.

5.4.Cross-Site Scripting (XSS)

An attack similar to SQL injection results from a web application failing to properly encode user-entered data when embedding it within Hypertext Markup Language (HTML) pages. Certain characters must be escaped when they are to be embedded within HTML, and failing to do so can allow malicious users to inject custom HTML and/or JavaScript code into data that is displayed by the web application. The injected code will then be interpreted and executed by the browser as any other HTML/JavaScript on the page would be; vulnerabilities such as these are called Cross-Site Scripting (XSS) vulnerabilities [4].

Consider a PHP page that contains the following code:

```
<form action="." method="post">
    <p><?php echo isset($_POST["data"])
? $_POST["data"] : ""; ?></p>
    <p><textarea name="data"></
textarea></p>
    <p><input type="submit" /></p>
</form>
```

If the user simply enters a string such as "hello", the string will be displayed above the form. The user may enter their own code, however; entering "hello" would cause the string to "hello" to be displayed in bold text. JavaScript could even be embedded into the page by entering something such as the following into the text area control:

```
<script>while(true)
window.alert("hello");</script>
```

This would cause a JavaScript alert window containing the string "hello" to be displayed repeatedly (some browsers now guard against such attacks by allowing the user to prevent a page from creating further alerts, as seen in Figure 4).

While displaying JavaScript alert windows is little more than an annoyance, serious attacks can be launched through XSS

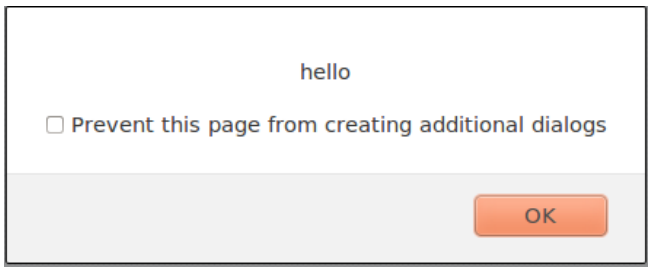

Figure 4. JavaScript alert window in Firefox.

vulnerabilities. JavaScript code can be used to access cookies, for example, which may contain sensitive user information. Code injected through an XSS vulnerability could easily access the contents of a cookie and send the data to a server under the control of an attacker.

XSS vulnerabilities occur out of a failure to escape characters that have special meaning in HTML, such as the less-than character that signifies the beginning of an HTML tag. These characters can be properly embedded within HTML pages by replacing them with HTML character references. A list of the most important character references can be found in Table 1.

Table 1. Commonly used HTML character references [20]

Character	Character Reference
<	<
>	>
&	&
"	"
Non-breaking space	

Strings can be properly HTML-encoded by replacing occurrences of the characters on the left side of Table 1 with the corresponding character references on the right side.

Server-side web programming languages and/or frameworks typically provide functions for encoding a string as described above. PHP's, for instance, is called htmlspecialchars, and it can be used in the second line of the code snippet above to prevent the user from embedding custom HTML into the page:

```
<p><?php echo isset($_POST["data"]) ?
htmlspecialchars($_POST["data"]) :
""; ?></p>
```

Some frameworks perform automatic encoding of data when embedding it in HTML output. The ASP.NET Razor engine for using C# code within HTML pages, for instance, includes a shortcut for outputting the values of variables within HTML: the "@" symbol can simply be prepended to a variable's name to embed the value of the variable within the page. This shortcut automatically performs HTML encoding of the value [9]. The developer should be familiar with the HTML encoding behavior of the programming language and framework being used to develop an application in order to avoid inadvertently embedding raw user input within HTML pages.

5.5.Proper Use of Hypertext Transfer Protocol (HTTP) Methods and Query Strings

The Hypertext Transfer Protocol (HTTP) query string is a popular mechanism for passing information from one page to another in web applications. While it's useful and easy to use, the developer of a web application should be sure to use it in an appropriate manner.

Google, for example, uses a query string parameter called "q" to pass in a string to search for. For example, one could open the following URL to search Google for "html5":

https://www.google.com/?q=html5

This is an appropriate use of the query string as the information being passed in is essentially a description of something that the user is looking for on the site. The query string should be used to retrieve information, but in general it should not be used to perform actions.

One common inappropriate use of the query string is to use it to pass user login information from one page to another. For example:

http://mywebapp/login?username=Admin&password=abc

The biggest problem with using the query string in this way is that query strings are generally included in web server logs and traffic analytics tools, so the username and password of each user that logs in would be logged in plaintext in web server traffic logs. If an attacker somehow gained access to these log files, they would have access to user login credentials.

Another issue is that the query string is displayed to the user in the web browser's address bar. A non-technical user may copy a link containing sensitive information in the query string and give it to someone else.

Finally, consider a query string that's used to post content in some way (for example, a blog web application that uses the query string to add posts to the blog). The URL using such a query string might look something like this:

http://mywebapp/addpost?title=My+Post&body=Hello

The above URL may be stored in the user's web browser history, and the user may inadvertently load the URL multiple times without realizing that the same content has been posted multiple times.

In general, the HTTP POST method should be used for performing actions (such as adding, deleting, and otherwise manipulating data) and passing sensitive user input into a page, while the HTTP GET method should be used solely for retrieving information [3].

5.6.Cross-Site Request Forgery (CSRF)

A class of attack closely related to the improper use of query strings is the Cross-Site Request Forgery (CSRF) attack.

A CSRF attack typically takes advantage of the fact that logging into a website may create a session that persists for a period of time; if a user has an active session on a site, an attacker may be able to cause that user to take certain actions without the user's knowledge [2].

Consider the example of the blog application mentioned in the previous section; if a user is logged into the application, an attacker may try to get the user to load a URL such as the following to have the user post a message of the attacker's choosing:

http://mywebapp/addpost?
title=Fraudulent+Post&body=This+message+was+posted
+through+a+CSRF+attack

The trick for the attacker is how to get the user to load such a URL. There are a number of ways to do this; one way to make it less obvious what's going on is for the attacker to create a web page that uses the above URL as the source of an image, and then get the user to visit the page [2]. Despite the fact that the URL may not return an image, an HTTP GET request is still made to the URL, and the message is posted without the user's knowledge.

Attacks such as the one described above can be prevented by not allowing such actions to be taken with simple HTTP GET requests; HTTP POSTs should be used instead when adding, deleting, or otherwise manipulating data. Although it's more difficult, it's possible for HTTP POSTs to be forged as well; one popular technique for preventing the forgery of HTTP POSTs is for the server to generate a unique token that's included in HTML forms and verified when an HTTP POST is received; if the token is not included with the request or is invalid, it can be assumed that a CSRF attack was attempted and the request can be discarded [2].

5.7.Use of Remote Resources

When creating a web application, the developer should be wary of using resources from remote servers. Examples of such resources would be images, style sheets, and JavaScript files that reside on servers that are not under the developer's control.

One obvious way in which this type of embedding of resources could be exploited is if the administrator of the remote server changed the content of the resource without informing the developer of the application making use of the resource. An image or style sheet could be replaced with something completely different from what the developer intended to appear on a web page, and malicious code could even be added to style sheets or JavaScript files stored on remote servers.

A less obvious way in which the embedding of remote resources could be exploited is if the administrator of the remote server started tracking the traffic to the application that uses the remote resources. Through the HTTP referrer header, the administrator of a server containing a remote resource can obtain details about how the application is being used, possibly even gaining sensitive user information if the query string is being improperly used as described earlier, since query string information is included in the HTTP referrer header.

If the developer does wish to use resources stored on remote servers, then he or she should ensure that the owner of the server is trustworthy before using such resources.

5.8.Securing Sensitive Files

The developer of a web application should be careful not to inadvertently place sensitive files in locations such that they are served by the web server software with no security mechanisms in place. In general, web applications have root directories that are served by the web server, and standard files (files other than PHP scripts, for example) may be served as-is when they are accessed through the web server.

Consider a web application that has a file sharing component, in which users can upload documents and send them to other users. It may be tempting to store uploaded documents in a location beneath the application's root directory that's served by the web server so that allowing the document's recipient to download the file is as simple as directing them to a location such as the following:

http://mywebapp/uploads/SecretDocument.doc

The problem with this, however, is that the web server does not know how to verify that the user accessing the document at the above URL is the intended recipient of the document. An attacker may be able to guess or otherwise determine the URL for a sensitive document that they wish to obtain and download the document without providing the appropriate credentials.

The solution to this problem is to store such documents outside of the application's root web directory, in a location that the web application can still read and write to. A script can then be created that will verify a user's identity using the web application's authentication mechanisms and write the document's contents to the HTTP response only if the user has access to the document. The URL to download a file using such a script might look like the following:

http://mywebapp/download.php?
file=SecretDocument.doc

If the user's identity is verified and it's determined that the user has access to the specified file, the script can set the response type using PHP's "header" function and use the "readfile" function to output the raw contents of the file to the HTTP response [16]. Note that the script should be sure to validate the filename given in the query string in order to ensure that a user does not try to read files other than those in the uploaded documents directory; for example, filenames should not be allowed to start with the "/ " character or contain the string ".." (which can be used to refer to the root directory and the directory one level higher than the current one, respectively). A better solution would to use numeric identifiers to identify documents (a file could be stored as 123.doc, for example), and that way the script can simply ensure that a valid integer has been passed in through the query string.

Another category of sensitive files that may be served through the web server are those which should not be accessible via the web application at all. A good example of this would be the files generated by the popular Git version control system; a developer may store an application's root web files in the root of the Git repository, which would result in the .git directory containing the entire version control history of the application being served by the web server [17]. A malicious user could potentially access these files by going to a URL such as the following:

http://mywebapp/.git/

This could result in an attacker gaining access to the source code for the application, which may not be desirable if the application is not open source software. The damage could be more severe if any sensitive information, such as passwords or encryption keys, have been committed to the Git repository.

The solution is to ensure that the application's root web files are not stored in the root directory of the Git repository; a subdirectory called "www" or similar could be created within the repository, and this subdirectory is what the web server would be configured to serve.

6.SOCIAL ENGINEERING

The final element of security to consider for web applications is the human element. A social engineering attack occurs when an attacker tricks a user of an application to perform some action or reveal some sort of sensitive information by the attacker pretending to be someone that he or she is not.

For the most part, social engineering attacks are outside of the control of a web application's developers, although some steps can be taken to reduce the risks of such attacks.

One step is simply to try to educate the application's users about such attacks. A common social engineering attack for web applications is for an attacker to request a user's password, claiming to be a system administrator or other authority figure associated with the site. It should be made clear to users that the application's administrators will never contact users asking for their passwords or other sensitive information. Additionally, if the application has some sort of builtin messaging mechanism to allow users to communicate with each other, the application should provide some sort of visual indicator to distinguish system administrators from normal users; this way, in the event that a system administrator does need to get in contact with a user for a legitimate purpose, the user can be assured that such communication is indeed coming from an administrator and not simply an attacker pretending to be one.

Education about social engineering attacks is particularly important for system administrators and others involved in the operation of a web-based service. They should be aware of common social engineering tactics in order to protect themselves and their organizations from such attacks.

Some technical steps can also be taken to reduce the risk of social engineering attacks. One possible attack would be for an attacker to trick a user into following a link that exploits improper query string usage or CSRF vulnerabilities in order to perform some action without the user's awareness. The solution is simply to try to minimize the occurrence of such vulnerabilities, using the mechanisms described earlier in this paper.

Other possible ways to mitigate social engineering attacks would be to limit the number of users that have administrative privileges (or other types of privileges that would make a user an attractive target for attackers) and to minimize the privileges of users that must perform administrative functions. In general, it's not possible to prevent social engineering attacks completely, and so the goal is to minimize the damage that can be done through a successful social engineering attack.

7.CONCLUSION

As the previous sections have shown, there are a number of factors involved in securing modern web applications. It's not just about writing secure code; it's also about choosing a secure operating system platform and keeping it up-to-date, choosing the appropriate HTTP and DBMS server software that provide the desired security features, and ensuring that everyone involved in the development and maintenance of a web application is aware of common vectors of attack.

The most rapidly evolving aspect of web application security is that involving technical attacks against the application itself. New variations on Cross-Site Scripting (XSS) attacks may emerge, and entirely new classes of attacks may develop as new technologies for web development are created and standardized.

Web application developers should be aware of the purposes of and the relationships between all the different parts of web applications — Hypertext Markup Language (HTML), JavaScript, Cascading Style Sheets (CSS), server-side code vs. client-side code, the HTTP server, the DBMS server, and more — in order to be aware of the possibilities and the security implications involved in using these technologies. Most technical vulnerabilities simply involve using these technologies in creative ways or ways in which the technology's creators did not anticipate. Understanding how these technologies work is the key to knowing how they can be exploited and being able to adapt quickly when new types of attacks emerge.

The breach of a web application can be catastrophic not only for the application's developers, but for its users. As such, the implementation of proper security practices is a critical aspect of web development, and everyone involved in the development of web applications should familiarize themselves with common attacks and countermeasures in order to minimize the chances of their applications being broken into.

8.REFERENCES

1. Apache Software Foundation. 2013. Apache HTTP Server Version 2.4 Documentation. http://httpd.apache.org/docs/2.4/.
2. Barth, A., Jackson, C., and Mitchell, J. C. 2008. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM conference on Computer and communications security, CCS 2008,* Alexandria, Virginia, October 2008, ACM, New York, NY, 75-88. DOI= http://dx.doi.org/10.1145/1455770.1455782
3. Berners-Lee, T., Fielding, R., Frystyk, H., Gettys, J., Leach, P., Masinter, L., and Mogul, J. 1999. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616. http://www.w3.org/Protocols/rfc2616/rfc2616.html.
4. CERT. 2000. Malicious HTML Tags Embedded in Client Web Requests. CA-2000-02. http://www.cert.org/advisories/CA-2000-02.html.
5. Community Ubuntu Documentation. 2013. Automatic Security Updates. https://help.ubuntu.com/community/AutomaticSecurityUpdates.
6. Cooper, P. 2011. rbenv: A Simple, New Ruby Version Management Tool. *Ruby Inside*. http://www.rubyinside.com/rbenv-a-simple-new-ruby-version-management-tool-5302.html.
7. Harrison, G. 2007. MySQL stored procedures: next big thing or relic of the past. *Linux Journal, 2007,* 164.
8. Henry-Stocker, S. 2005. Running SSH on a non-standard port. *ITworld*. http://www.itworld.com/nls_unixssh0500506.
9. Microsoft. 2012. Introduction to ASP.NET Web Programming Using the Razor Syntax. http://www.asp.net/web-pages/tutorials/basics/2-introduction-to-asp-net-web-programming-using-the-razor-syntax.
10. Mitchell, S. 2009. Exception Handling Advice for ASP.NET Web Applications. *4 Guys From Rolla*. http://www.4guysfromrolla.com/articles/081209-1.aspx.
11. Moses, P. 2006. Demons Seeking Daemons—A Practical Approach to Hardening Your OpenSSH Configuration. *Linux Journal, 2006,* 143, 3.
12. OpenSSL. 2013. ciphers(1). http://www.openssl.org/docs/apps/ciphers.html.
13. Oracle Corporation. 2013. MySQL 5.5 Reference Manual. http://dev.mysql.com/doc/refman/5.5/en/.
14. Paradis, P. 2009. How to Make a Self-Signed SSL Certificate. *Linode Library*. https://library.linode.com/security/ssl-certificates/self-signed.
15. Paradis, P. 2009. Obtaining a Commercial SSL Certificate. *Linode Library*. https://library.linode.com/security/ssl-certificates/commercial.
16. PHP. 2013. Runtime Configuration. PHP Manual. http://www.php.net/manual/en/errorfunc.configuration.php.
17. Python Sweetness. 2013. Devs, please stop serving .git to the outside world. http://pythonsweetness.tumblr.com/post/52587443706/devs-please-stop-serving-git-to-the-outside-world.
18. Russell, C. 2008. Bridging the Object-Relational Divide. *ACM Queue, 6,* 3, 16-26. DOI= http://dx.doi.org/10.1145/1394127.1394139
19. SQLite. 2013. SQLite is Self-Contained. http://sqlite.org/selfcontained.html.
20. World Wide Web Consortium. 2013. The HTML syntax. HTML5. http://www.w3.org/TR/html5/syntax.html.
21. World Wide Web Consortium. 1995. Privileged Ports. http://www.w3.org/Daemon/User/Installation/PrivilegedPorts.html.