Deploy Secure and Safe Web Applications

Part of the PHP on IBM i series

Deploy Secure and Safe Web Applications

By kind permission of BCD Software

The purpose of this white paper is to explain how WebSmart, in conjunction with the free IBM i HTTP Server powered by Apache, provides all the tools and technology for you to create and deploy web applications that are totally secure and safe. These applications can be browser-based or SOA.

Areas of concern

There are six major areas of concern with regards to security:

- Client (browser)
- Network connection (internet, intranet, extranet)
- Firewall
- HTTP Server
- WebSmart applications running on the IBM i (server)
- SOA applications - requiring communications between an IBM i server and any other server using the HTTP or HTTPS protocols.

We refer to each of these areas of concern in the following sections, and the security issues specific to each one.

Security issues

This guide will address the following security issues:

- Saving information on a client - using cookies
- Preventing XSS (cross-site scripting) and SQL injection attacks
- Sending secure information back and forth from client to server using encryption
- Saving secure information on the server using encryption
- Prohibiting unauthorised access to the web server
- Prohibiting unauthorised access to OS/400 objects (programs, files, data areas, libraries, etc.)
- Preventing DOS (denial of service) or DDOS (distributed denial of service) attacks such as ping floods or HTTP request floods
- In SOA applications, sending secure information back and forth between servers (web services).

We will explain the network, HTTP, IBM i system architecture and WebSmart programming features that make it possible to effectively address all these issues.

Saving information on a Client: using cookies

In this discussion, the term 'client' refers to a browser, running on any platform such as Windows, Macintosh or Linux. Most browsers and PC's have the capability to store persistent data on the PC, in the form of 'cookies'. A cookie is a small file containing information related to a specific site. Browser security (for all browsers) is designed so that a cookie created on one site cannot be accessed, modified or recreated by another site (so as to prevent the ability to hijack someone's cookie from another site.)

WebSmart has functions that fully support the use of cookies. While most browsers provide the ability for users to prevent cookies from being created or stored, you should understand that most commercial sites depend upon cookies in order to function correctly. Cookies often contain human-readable data, though, so when you decide to store information in them, that information should be of a non-sensitive nature. This will prevent other users of a client machine from having access to any sensitive information.

WebSmart provides functions for creating and managing sessions, called 'Smurfs' in WebSmart ILE and sessions in WebSmart PHP. You can use these functions to store a session ID in a cookie. This is simply a string of characters- 32 unencrypted, 64 encrypted. A smurf or session variable can be used to store sensitive information associated with it on the server instead of the client. We discuss this notion in more detail in a later section titled **Saving secure information on the server using encryption.**

Preventing XSS (cross-site scripting) and SQL injection attacks

Cross-site scripting is a security vulnerability that can enable an attacker to inject a client-side script into your web application. This type of attack may be used to bypass security mechanisms and validation normally imposed on web content to gain access to session cookies and other information kept on the client side or browser. In essence, XSS is a special case of code injection. The first line of defence against these type of attacks is contextual output encoding or escaping. There are various escaping schemes that can be used, including HTML entity encoding, JavaScript and CSS escaping and URL encoding. WebSmart PHP templates generate a function that will "sanitize" the output as part of these prevention steps. Besides content filtering, the use of additional security controls when working with cookie based authentication is recommended.

SQL injection attacks are another type of code injection techniques. SQL injection is a security vulnerability in which an attacker will input SQL statements in a web form to try and change database table contents. This could lead to an attacker gaining access to sensitive information stored in your files. The vulnerability happens when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and unexpectedly executed.

The article **Common Security Mistakes in Web Applications** hosted by Smashing Magazine; is an excellent starting point to gain a better understanding of these topics and other security concerns. This article contains straightforward examples and a solid explanation of each concept. Even though the article refers to PHP coding techniques, the same concepts apply for web applications developed with other programming languages or tools, including WebSmart ILE.

Sending secure information between Client and Server

While data can be secured readily by using encryption and conventional AS/400 object security, there will often be applications where you need to secure data as it is being transmitted across the Internet. For example, any time a credit card number is required to purchase a product, that number should be transmitted securely. This is where SSL (secure sockets layer) technology comes into play.

SSL is a protocol and mechanism that ensures that data is encrypted at the source, transmitted in encrypted form, and then decrypted at the target. The server or browser can play the role of source or target, depending on the nature of the transaction. For example, sending a credit card number for a purchase involves the browser as the initiator of a message sent to the server. The encryption technology used involves private / public key encryption. The server must have a digital certificate installed. This is essentially the 'public' part of the key. While the mechanics of private / public key encryption are beyond the scope of this guide, it is sufficient to know that all web servers (IBM i or not) that implement SSL require a digital certificate.

The IBM i has, as part of OS/400, facilities for creating your own digital certificates. However, we recommend that you purchase a digital certificate from a trusted CA (certificate authority) such as Verisign or Thawte, so that anyone who accesses the secure part of your site will know that the certificate has been authenticated by a reliable third party, as opposed to one you have simply generated yourself. Bear in mind that implementing SSL on the IBM i is a process completely independent of using WebSmart. Any static pages, CGI programs, or Java servlets can use the SSL infrastructure on your server once it has been configured and activated.

To specify the use of a secure channel, with SSL in effect, you can use https:// in your URL (as opposed to simple 'http://'). This informs the browser that the link is secured. For example, a shopping cart program called CART that requires a credit card number could be invoked by clicking on a link to a URL such as the following (non-working example link only):

https://www.bigcompany.com/cart.pgm

Clicking on this link will result in communications between browser and server utilizing SSL. All information sent back and forth between them will be encrypted at the source, transmitted in encrypted form, and then decrypted at the target, thus preventing any bots from sniffing and reading sensitive information.

For additional resources on implementing SSL, please refer to the WebSmart Reference Guide, Chapter 10, Security. IBM also has reference manuals and Redbooks on the subject. Search for SSL in the IBM i information centre, accessible from **www.ibm.com**.

Saving secure information on the server using encryption

Once information has been transmitted securely, via SSL, you need a way to store that information securely. While conventional IBM i object-level security can be implemented (discussed in a later section), you may need additional security. Credit card numbers and passwords are examples of data that is so sensitive it requires some additional security.

You can use the encryption routines provided with WebSmart to store just those pieces of information that are especially sensitive in encrypted form. For example, a customer master file might contain a number of fields that do not need to be private, such as name, address, city etc., while data elements such as password and credit card need to be encrypted. You can use the encrypt PML function to encrypt data and the decrypt PML function to decrypt it. These functions use AES encryption, which utilizes a seed encryption key. As an additional security measure, you will probably want to secure the encryption key from programmers (secure any source code by deleting it or revoking authority to it) to ensure no programmer can simply decrypt data by calling the decrypt function using the correct key.

If you use this approach, even if a user or programmer can view the contents of a file using some kind of IBM i utility (DFU, SQL, DSPPFM etc.) or program, the sensitive data elements will be unintelligible.

In an earlier section we discussed using session ids (smurf ids) to associate a given user's browser session and interaction with the server. Session ids provide a convenient programming mechanism for maintaining state in a web application. They also allow you to store sensitive information associated with a session in smurfs, which are essentially server-side cookies. A session variable is simply a piece of data stored in a protected database file in WebSmart. It is associated with its unique smurf (session) id. So, even if you don't choose to store sensitive information such as credit card numbers in your legacy application database, you can use a smurf to store it instead- either for a very short time (such as the life of the browser session), or as persistent data. You can use the encryption functions to encrypt and decrypt data stored in smurfs in the same manner as for conventional database fields.

Prohibiting unauthorised access to the web server

Contrary to popular misconceptions, the IBM i Web Server is actually considerably more secure, by default, than the library system of the IBM i. In the library system, objects are implicitly publicly available until you explicitly revoke authority, or assign custom authority (use and management rights). Entire libraries can be left unsecured, with the only line of defense being the inaccessibility of a command-line interface. In contrast to this, the HTTP original server, when running with the default configuration, denies access to everything; so does Apache. For example:

```
<Directory />
AllowOverride None
Options None
Order deny,allow
Deny from all
</Directory>
```

This configuration code tells the Apache web server to deny access to all directories. You can then modify the configuration file so that subsequent directives selectively open up parts of your server. For example:

```
# Tell Apache where to find programs from
url /webtest/ ScriptAliasMatch
^/webtest/(.*)\.pgm$
/qsys.lib/webtest.lib/$1.pgm

# Allow all programs in this directory to
run:
<Directory /qsys.lib/webtest.lib>
AllowOverride None
Options None
order allow,deny
allow from all
</Directory>
```

These directives allow CGI programs in library webtest to run. Note that you have to explicitly tell Apache which libraries are available to run programs from.

How to challenge requests for valid user IDs and passwords

In addition to opening access to very specific, limited areas of the server, you can also cause the browser to issue challenges for user authentication. This technique allows you to ensure that only authorised users access those areas of the server that have been opened up. You can base user authentication on IBM i user ids, in which case all IBM i authority features are invoked, including the '3 mistakes and you are disabled' feature for repeat wrong password attempts. Or, you can use validation lists. A validation list is an OS/400 object of type *VLDL. It contains entries with both secure and unsecured information (e.g. user ids and passwords).

When the browser challenges a user for their user id and password, the HTTP server can validate that information against a validation list entry. Another option for user authentication is to use a database file, and authenticate with a WebSmart program that validates the user and password against records in that file. In this case, you probably want to secure passwords by storing them in encrypted form in the file (using the AES encryption functionality of WebSmart). Using this approach requires that you write a WebSmart front-end login program, and that all subsequent programs in the application check to ensure that the user has successfully logged in. You can do this with the session id functions of WebSmart. For example, the login program can do the following:

- Prompt for user id and password
- Validate against a database file (match the user id by key, and match on decrypted password)
- When validated, create a session id (smurf id) and set a smurf (server-side cookie) to store the user name, and any custom authority settings.

Subsequent programs would do the following:

- Check for the presence of a valid session id
- Check for the presence of a smurf for that session id, containing the appropriate value to indicate the user is authorised to proceed
- If no session id or smurf is found, redirect to a 'not authorised' page.

Prohibiting unauthorised access to OS/400 objects

By using the HTTP server authentication method for protecting access to various areas of your site, you can also impose standard OS/400 object-level security. Normally, a special user profile of QTMHHTP1 is used for jobs running in the web server.

Object rights will be determined by the authority to each object available for QTMHHTP1. As an example of how to impose object-level security, let's assume you have a common library of files, used by both green-screen users and by some web users.

If you want to ensure that only green-screen users with the correct authority can access certain files in the application, you can either explicitly grant data rights to files for those users, or grant rights to *PUBLIC and explicitly revoke rights for user QTMHHTP1. The web server protection directives also let you specify any other user profiles to run under, in addition to the default of QTMHHTP1. In the original HTTP server, these are protect directives.

Adopting user profiles other than QTMHHTP1

A protect directive in the HTTP server configuration can specify to use the default user profile (QTMHHTP1), or the user profile of the user who has signed on, just like in a traditional 5250 interactive job. In the following example, whenever a user tries to access a page with /cgidev/ in the path portion of the url, they will first see a challenge box, asking for a valid user id and password. That request, and all subsequent requests to that path from that client, will use the user profile that is entered.

```
Protection ProtMySite
{
  ServerID MySite
       PasswdFile %%SYSTEM%%
       ACLOverride Off
       mask All
       UserID %%CLIENT%%
}
  protect /cgidev/* ProtMySite
  exec /cgidev/*.pgm
  /qsys.lib/cgidevpgm.lib/*.pgm
  pass /cgidev/* /compweb/dev/*
```

So, for example, if the user attempts to go to this url: www.bigcompany/cgidev/login.pgm the browser will challenge them for a valid user id and password. Because %%CLIENT%% (instead of %%SERVER%%) is specified for the UserID keyword in the protection directive, the web server will handle the request using the user profile specified at the client. So, if the user signs on as FRED, then all object rights, etc. will be checked against FRED's user profile, not QTMHHTP1.

You can also configure the Apache web server to accomplish the same thing. Here's an equivalent example:

```
<Location /cgidev/>
Require valid-user
AuthType Basic
AuthName ProtMySite
PasswdFile %%SYSTEM%%
UserID %%CLIENT%%
</Location>
```

You can also adopt a user profile directly in a WebSmart program. WebSmart comes with a set of PML functions for using IBM i user profiles. These functions provide all the capabilities you have in interactive jobs - validating the user profile, changing passwords (if you have authority to do so) and causing a job to run under a given user profile. They are implemented by using native OS/400 APIs that work with user profile security at the operating system level.

These PML functions provide the additional flexibility to control security at a program interface level, instead of at the HTTP server level. For example, you could write a WebSmart program that presents an initial login page, asking for IBM i user profile and password. Once the user has successfully entered a valid user profile and password, any subsequent page requests will use this for security purposes. For example, if user FRED logs on, and attempts to access the payroll file, but does not have read data rights to the file, then IBM i object-level authority will kick in and prevent FRED from accessing the data. Object-level authority applies to any OS/400 object type, such as data areas, user spaces, spool files and programs, not just database files.

In summary, you can allow or deny access to OS/400 objects using these mechanisms:

- HTTP Server configuration directives
- WebSmart user profile functions and coding techniques that interface with native OS/400 APIs
- OS/400 object-level security

Imposing additional security by extending the web servers

Both the Original HTTP Server and the IBM HTTP Server Powered By Apache can be extended with user-written code to allow you to impose additional security constraints on your server. Normally, all URL requests are handled by the web servers configuration directives, as discussed in a section above titled Prohibiting unauthorised Access to the Web Server. As a request comes in to the server, it is picked up by the HTTP server, and matched against the configuration rules to see if it qualifies for consideration for a specific action. If it does, that action is taken. This might be to explicitly allow or deny access to that file, or to explicitly running a program. For example, in the Original HTTP server, an exec statement is used to control which CGI programs can run, while a pass statement determines what static files can be served. There are other types of statement, such as service statements, that allow you to specify a module of code to run for a URL. For example:

```
service/nexus/intra/*.pgm/QSYS.LIB/XL_SMSLIB
.LIB/sc_httpsrv.srvpgm:xl_service
```

Here, any requests such as:
`http://myserver/nexus/intra/nxmenu.pgm` will be intercepted by this statement, and passed to the service program call `sc_httpsrv in XL_SMSLIB`. This service program needs to be coded to conform to the requirements for receiving URL requests within the HTTP server. For example, in Nexus, our IBM i-centric Web Portal product, we use this service program to further qualify program execution requests. Using this approach, we can control which specific Nexus users are authorized to view pages or run Nexus components. This allows us to have a database-driven security system that provides an extension to the existing security constraints, much like a 5250 menu system can provide additional security to standard OS/400 object authority.

In this case, the Nexus web server extensions check against a links protection database that determines if users or groups of Nexus users are permitted or denied access to programs or static pages. Because this security measure takes place within the web server, it is extremely secure- nothing has to be done at the application programming level, or the operating system object security level, because unauthorised requests will be rejected by the HTTP server itself, prior to being conveyed to operating system.

Imposing additional security by running the web servers separately

There are several advantages to running the database and the web server on a different server and/or partition.

The most important advantage is the added security. This is assuming that your database and web server are behind a firewall and have only minimal links open to each other. If one of the servers is compromised, the maximum damage the other one will get is the link/API between the two servers. Compare this to hosting the web server and database on the

same box. If the web application is compromised, the hacker could potentially access the server files. They would also have access to the entire system, including the database.

Running the web server seperately also allows for better scalability. For example, if you need more web server performance, you can add another web server into the cluster. In this case, there would be two web servers handling the load, which would offload some of the web requests from the existing web server onto the new one.

This example leads to another advantage - redundancy. If you have two or more web servers running, you can potentially take a web server offline to perform upgrades and let the other web server handle the requests. This would allow you to complete upgrades while your site is online. Once the upgrade is done, you can easily put the upgraded server back to work.

Ping floods, DOS or DDOS floods

This subject is really concerned with general network security issues, rather than specific to WebSmart and the IBM i, but we discuss it here because, regardless of what platform you choose for your web server, you need to give consideration to how you will prevent attempted attacks on your network from adversely affecting your operations.

Ping, DOS or DDOS floods are attacks on your server where your server is bombarded with incoming requests for responses. DDOS attacks are more lethal than DOS attacks, because many machines are involved in bombarding your server. These kinds of attacks are executed by malicious internet users. Some are sophisticated programmers, some are 'script-kiddies' - people with some technical knowledge who know how to run a pre-programmed script. The premise behind all these types of attacks is the same: flood your server with so much traffic that it eventually locks up.

The best approach to use to mitigate the effects of these attacks is to use a sacrificial lamb - a combination of hardware/software that absorbs the hits as they occur, but which does not allow any of the flooding traffic to go through to the web server or other servers (mail, FTP, file servers, etc.) inside your network. This is one of the roles of firewalls and routers. So, in general, it is a good idea to have a firewall in place at the conjunction of the internet to your internal network. The firewall may end up going down, but it will prevent any other part of your network from being affected. Although it is beyond the scope of this document to cover them in detail, there are many commercial solutions available to cope with these sorts of attacks. Some companies address this by having an IBM i in their DMZ exclusively dedicated to web serving, which accesses data on their actual production IBM i. In every case it is recommended that a firewall be used as well.

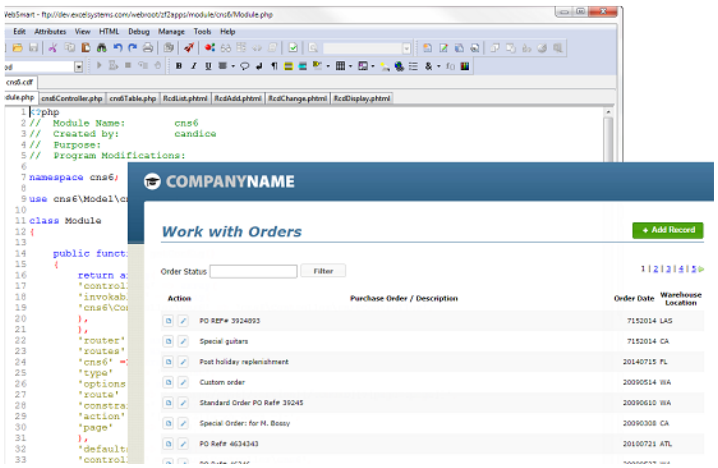
Smaller scale DOS attacks can be mitigated by controlling the number of threads (jobs) that the web server will spawn under heave loads, and to limit the priority of the web serving jobs. There are also a variety of configuration options within the Apache configuration which relate to detecting and handling DOS attacks.

SOA applications such as web services

SOA (Service Oriented Architecture) applications usually involve web services - the transmission of information directly between servers over the HTTP transport protocol. An example of an SOA application is credit card validation and approval. In this case, a server sends credit card information along with a potential sales transaction to a web server, in the form of an HTTP request. The initiator of the request is referred to as a web service 'consumer' while the web server that responds and provides the validation data is a web service 'provider'.

In order for such a transaction to be secure, the data must be encrypted at either end of the connection and transported back and forth in encrypted form. This is done by using the HTTPS (HTTP secure) protocol. WebSmart fully supports acting as both a web services consumer provider and consumer using HTTPS. So, in the above example, you can write a WebSmart application that makes a request to a credit card approval web server - your application acts as the consumer - and send that request totally encrypted. In addition, the response can be received across an encrypted channel, ensuring no-one can 'sniff' the data and read it as it transits the public network of the internet. Furthermore, as with browser-based applications, once any response is received by the WebSmart application, you can encrypt it using AES encryption, as described earlier, to store it on the server in an undecipherable form.

Note that SOA applications are platform-agnostic, meaning that as long as you have two web servers that fully support HTTP and HTTPS, it doesn't matter what the hardware, operating system or web server software (eg Apache, IIS) is.



Conclusion

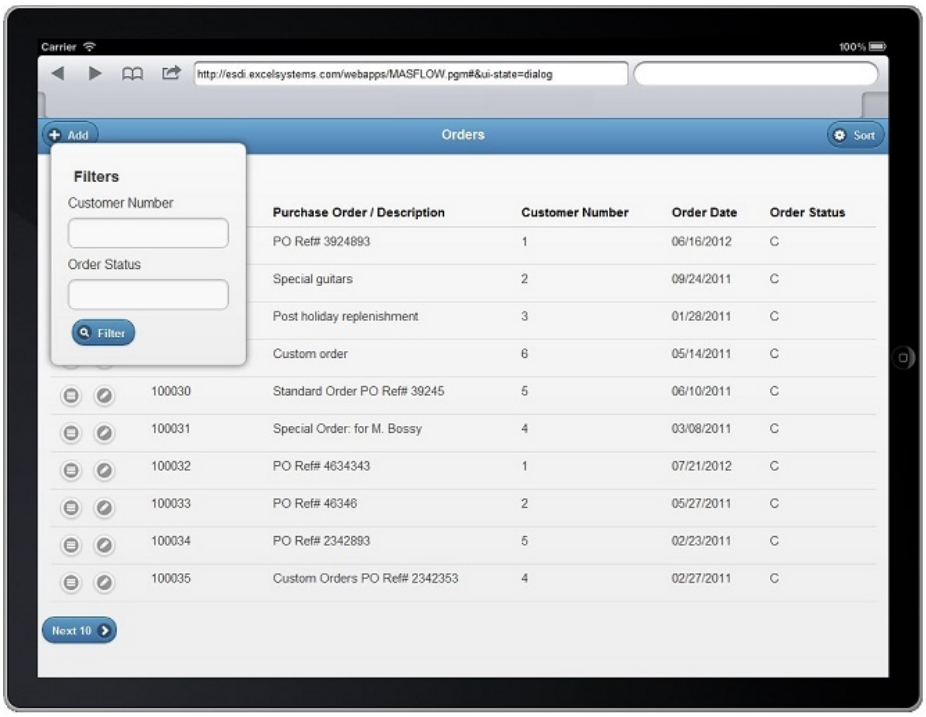
Security issues on the IBM i for web applications are comprehensively addressed by the following combination of technologies:

- SSL HTTP transport mechanism between client and server (where the 'client' is a browser)
- SSL HTTP transport mechanism for SOA applications implemented as web services
- Firewalls and routers
- IBM i-centric web server software: Original HTTP server or IBM HTTP server powered by Apache
- WebSmart 128-bit AES encryption algorithms (included in WebSmart)
- OS/400 object-level security

By using each of these technologies in the appropriate manner, as outlined in this document, you can develop and deploy IBM i-centric database web applications that are extremely secure, safe and reliable. These can be either browser-based applications or SOA applications.

*Guide reproduced with kind permission of **BCD Software**, who provide easy-to-use, affordable web application development and modernisation solutions that improve developer and end user productivity.*

*Follow the link to find out more about **WebSmart, the rapid development tool for creating PHP or RPG desktop and mobile applications**.*



Useful links and information about PHP on i

Zend is the leading provider of enterprise-grade applications for PHP. The PHP engine for IBM i, Zend Server, will run natively on your IBM i and is available as a simple and free download to all IBM i users.

BCD Software has been supporting PHP on IBM i since it first launched a specialised PHP version of WebSmart in 2007. Proximity is the exclusive UK and Ireland Partner for BCD Software.

PHP is the world's most popular programming language for web and mobile applications.

Proximity

For information about PHP on i, visit www.proximity.co.uk/resources/php-on-i

Partnering with some of the world's foremost software companies, Proximity develops, delivers, maintains and supports high performance solutions and applications for leading global companies in the logistics, manufacturing, retail and finance sectors.

Part of the PHP on IBM i series

For information about PHP on i, visit www.proximity.co.uk/resources/php-on-i

Proximity Group

t: +44 (0) 113 393 3360

e: info@proximity.co.uk

Leeds office

4-6 Kerry Hill, Horsforth,

Leeds, LS18 4AY

Nottingham office

Pure Offices, Lakeview Drive,

Nottingham, NG15 0DT