Web Application Security

TISC 2000

Eran Reshef, Founder
and
Izhar Bar-Gad, CTO
Sanctum Inc.

Abstract

Providing Web Application Security for an eBusiness is a huge and complex task. Every entry point in the e-Business system must be secured, at both the network and application levels. Whereas most network security issues, including access control, data transmission security, and authentication can be addressed using commercially available products, application security has received less attention. Consequently, the application has remained the most vulnerable component in the security chain. Today, almost every e-Business web site can be broken into at the application level in a matter of hours. Hacking techniques such as hidden field manipulation, parameter tampering, and cookie poisoning can be easily deployed, resulting in access to intellectual property and corporate assets; stolen customer data; alteration of prices; and defacing or debilitating the site to completely shut-down the site. The vulnerabilities that permit these exploitations exist as a result of flaws in the design, implementation, and testing of internally developed code, as well as bugs and misconfigurations of third-party products. Attempting to plug all these holes requires a full-time security team to monitor and patch the application.

In this paper, we describe a new security technology, Automated Web Application Control and Security (AppShield), a run-time application-level security proxy that automatically recognizes the application security policy for each page by constantly analyzing the outgoing traffic from the web application to its clients. The proxy then automatically enforces this policy on returning requests, preventing hackers from exploiting application vulnerabilities and removing the need to track and patch every hole in the application. This not only provides a higher level of security, but also reduces security resource requirements within the organization.

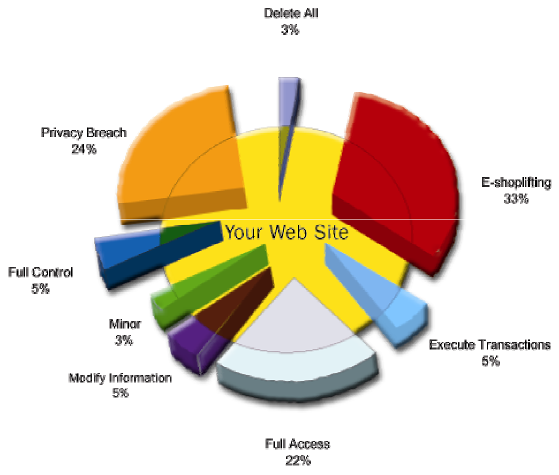
<u>Table of Contents</u>

Web Application Security - The Missing Piece

Audits performed on over 50 major web sites revealed that all of them had major problems at the application level that could be exploited in a matter of hours. While heavily secured at the network level, these sites still allowed hackers to execute Unix shell commands, download source code and even submit SQL queries via web application vulnerabilities. Why? Because virtually all sites today attempt to achieve application-level security manually. This is a complex task, whose final goal is to ensure that web applications interact with end users only in ways that were intended by the developers. Manual security measures include fortification of the application and its environment and recurring tests of the application and all third party applications. On the way to this ambitious goal, all web site managers struggle with the same issues:

1. Flaws with the design, implementation and testing of internally developed code, such as those found in Microsoft's Hotmail and other sites.
2. Vulnerabilities found in vendor products used to provide application infrastructure, such as web servers and application servers. More than 20 vulnerabilities were found in Microsoft's web server in 1999 alone. These are documented in Securityfocus.com.
3. Misconfiguration of underlying infrastructure, such as enabling of server-side-includes in web servers, or even allowing directory browsing (Click here to see Apache's security tips for server configuration).
4. Flaws with code obtained from external sources or with code that is being outsourced, such as shopping cart CGIs that store price information within hidden fields. (Click here to search AltaVista for examples.)
5. Backdoors and debug options left open on purpose within the application. For example, Matt's formmail.cgi, a generic WWW form to an e-mail gateway, can be used to pilfer the environment variables by using a debug flag ("env_report") and changing the recipient parameter.

The results of the audits are shown in the following graph, divided according to the outcome of the audit process:



Application Hacking Techniques

While it is very hard for web sites to secure their applications, application hacking is quite simple. A hacker typically spends a few hours understanding the web application, thinking like a programmer and identifying the shortcuts he would have created if had he built this web application. After doing so, the hacker uses a web browser and attempts to interact with the application and its surrounding infrastructure in malicious ways.

To better understand how easy web application hacking is, let’s look at three simple techniques:

Hidden Manipulation

Hidden fields are often used to save information about the client's session, eliminating the need to maintain a complex database on the server side. A client does not normally see the hidden field and does not attempt to change it. However, modifying form fields is very simple. For example, let's assume the price of a product is kept in a hidden field, a common practice allowing for e-shoplifting, and thus is trusted by any back-end system. A hacker can change the price, and the invoked CGI will charge him/her for the new amount, as follows:

1. Open the html page within an HTML editor.
2. Locate the hidden field (e.g., "`<type=hidden name=price value=99.95>`")
3. Modify its content to a different value (e.g. "`<type=hidden name=price value=`**`1.00`**`>`")
4. Save the html file locally and browse it.
5. Click the "buy" button to perform electronic shoplifting via hidden manipulation.

Parameter Tampering

Failure to confirm the correctness of CGI parameters embedded inside a hyperlink can be easily used to break the site security. For example, let's take a search CGI that accepts a `template` parameter:

```
Search.exe?template=result.html&q=security
```

By replacing the `template` parameter, a hacker can obtain access to any file he wants, such as `/etc/passwd` or the site's private key, e.g.:

```
Search.exe?template=/etc/passwd&q=security
```

Cookie Poisoning

Many web applications use cookies in order to save information (user id, time stamp, etc.) on the client's machine. For example, when a user logs into many sites, a login CGI validates his user name and password and sets a cookie with his numerical identifier. When the user checks his preferences later, another CGI (say, `preferences.asp`) retrieves the cookie and displays the user information records of the corresponding user. Since cookies are not always cryptographically secure, a hacker can modify them by modifying the cookie file, thus causing the return of information belonging to another user and enabling the performance of activity on behalf of that user.

<u>Manual Application Security</u>

To manually subvert these attacks, as well as others, a web site development team needs to go through a cyclic process that spans the entire organization and exacts a toll in each phase of web site management.

Secure code design

Designing application functionality with security in mind leads to a more complex application and extends development time. In addition, designing a secure application requires specific expertise that may not be available within the organization. In addition, any major change in the site will force re-examination of the design.

Secure code implementation

Implementing a secure application requires the use of defensive coding, i.e., embedding checks and balances, to make sure an implementation error will not cause a security hazard. A more complex design also complicates implementation demanding more time for coding and testing. Sparing time for such tasks is usually a luxury that is unavailable in the rapidly changing world of web development. Some application servers can provide limited assistance in this area although none of them can supply a complete solution.

Testing for Loopholes

Other than functionality testing, an entirely new category of stress testing must be implemented. The application should be placed in hostile environments and attacked with various tests and inputs designed to expose its loopholes. This security testing process demands expertise, which do not necessarily exist within the development or quality assurance groups of the organization causing a need for expensive outsourcing.

Secure Configuration

Careful attention to detail is crucial in this stage, as the configuration of each component should be checked and verified to disallow any exploit. This includes web servers, application servers, public-domain CGIs and, of course, internally developed code. For example, the site administrator should configure vendor software to turn off any unsafe features, set correct permissions on every file that is accessible by the web servers, remove debug and features and options left for the quality assurance process from production environment and remove default examples. When using hardened web servers, secure configuration is easier to achieve than with normal web servers.

Constant Patching

Every time a vendor or a public-domain CGI developer announces a fix for a vulnerability found, the patch should promptly be applied to the entire site. It is very hard to keep pace with the rate of the fixes, especially for large, complex sites.

Education

Educating developers, testers, site administrators and external consultants to understand and master application security is a daunting task. You will always have some novice people who are bound to make mistakes.

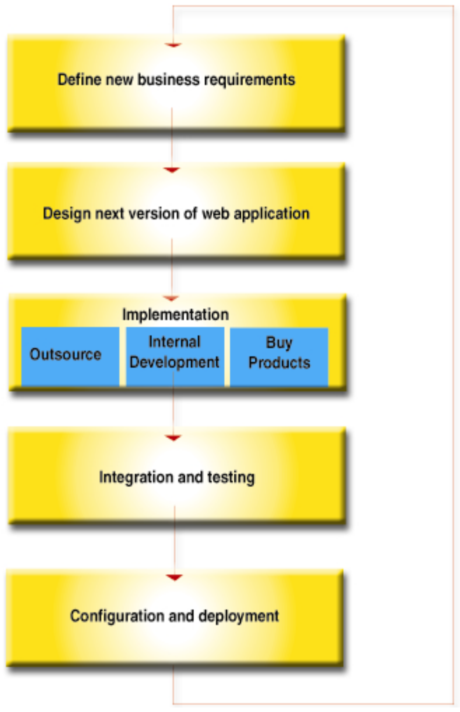
Code Reviews

Public-domain software is widely spread in the web environment, this software usually contains security holes that are easily examined by hackers. A code review is normally needed to ensure its security properties. This code review is a very time intensive process that must be on going to deal with the constant advances in the software. Moreover, code reviews might be needed for the software developed in the organization to find backdoors left by the application programmers.. The only way to remove these in-house backdoors is to have a third-party advisor review all your code, a costly and time consuming process.

Why Manual Application Security Fails

Unfortunately, all manual application fortification fails in the long run. This is due to the complexity of the product and the fact that it is constantly changing.

Life cycle of a typical web based application


Define new business requirements
Design next version of web application
Implementation
Outsource
Internal Development
Buy Products
Integration and testing
Configuration and deployment

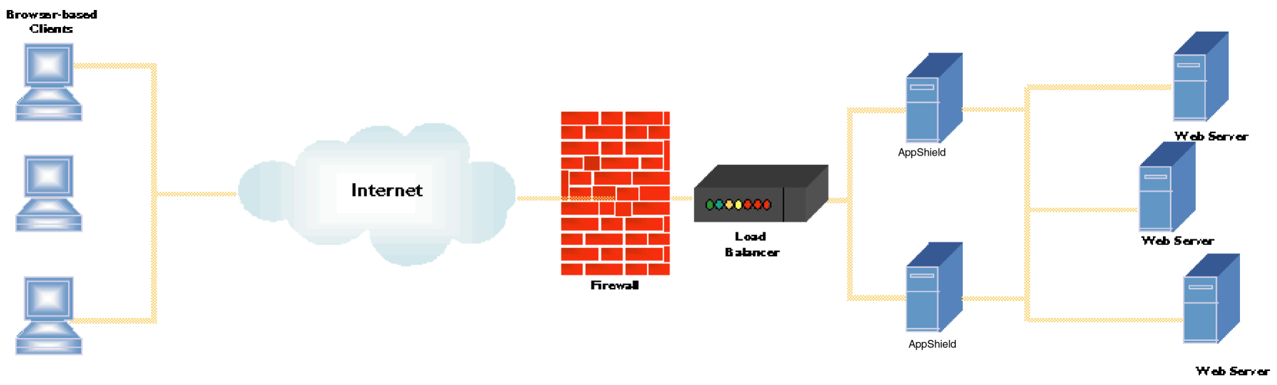
Since there are multiple steps, an error leading to a security vulnerability might occur in any of the stages and affect the whole sequence. A single design stage performed in an insecure way is enough to cause the application to be insecure even with the best implementation. Furthermore, since the sequence of stages occur in "Internet time," new security holes are likely to pop up quite often. Even assuming that at each stage there is a mere 1% chance of a mistake, after iterating through these stages several times the chances are multiplied and a security vulnerability becomes highly probable. For example, a system administrator might remember to add all the patches to his web servers. However, when he adds a new web server a year later he may not remember to add the old patches (e.g., the hack into MailStart site). This continuous cycle of trying to keep up with all the patches led to the following comment from Yahoo after their site was hacked: *"It would be naïve to promise that there'll be no bugs in the future*[1]".

[1] http://news.cnet.com/news/0-1007-200-340937.html

AppShield - A Solution to the Web Application Security Problem

Instead of trying to patch all the holes – an impossible task – there is a way to secure the application:simply refuse to allow hackers to exploit the vulnerabilities. AppShield automatically secures web applications on the fly. As HTML pages are requested from a web server to a browser, *AppShield* automatically generates a security policy tailored for the web application. The *AppShield* process automatically extracts all of the acceptable responses defined in the HTML page**,** and enforces HTTP requests to conform to the automatically generated security policy when they return from the web browser to the server. *AppShield* resides between the Internet and the application, usually behind firewalls and load balancers and in front of the web servers, where it functions like a proxy for bi-directional information flow of requests and responses.

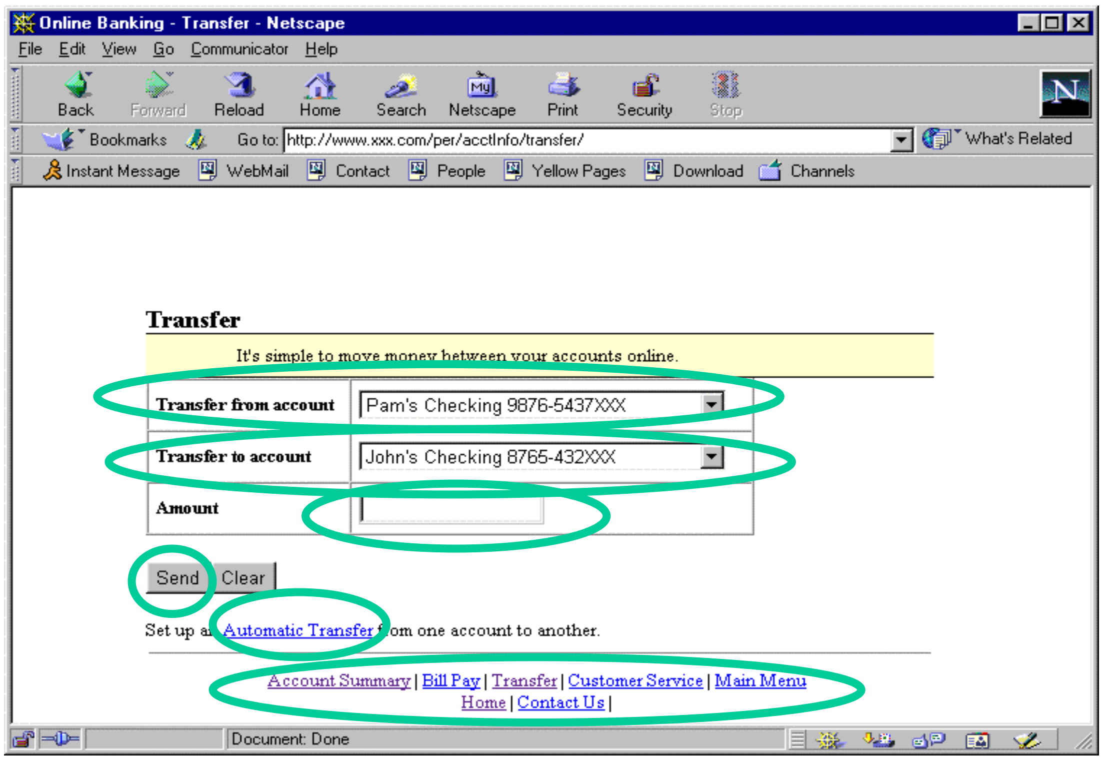
AppShield within the organization network.



When a user starts an application session by directing his browser to an e-business site, *AppShield* first verifies that the page accessed is indeed a legal *entry URL* to the site. For example, the site administrator may declare the home page to be a legal *entry URL* as well as any page under the “products” section. After the initial check is performed, *AppShield* creates an *application session token* and stores it inside a cookie that is cryptographically protected by AppShield. This cookie is used in all future transactions to uniquely identify users.

Once a session is established, *AppShield* analyzes each HTML page that belongs to that session as it is being forwarded to the browser. The *Policy Recognition Engine* analyzes the page, looking for information such as CGI parameters, hidden field values, drop-down menu values, and maximum size of expected text fields. Based upon this run-time analysis, *AppShield* automatically determines the security policy of the application. Additional legal requests cause *AppShield* to adjust the security policy for the session.

AppShield's Policy Recognition Engine automatically identifies the security policy of each HTML page.



Online Banking - Transfer - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks
Go to: http://www.xxx.com/per/acctInfo/transfer/
What's Related
Instant Message WebMail Contact People Yellow Pages Download Channels
Send Clear
Account Summary | Bill Pay | Transfer | Customer Service | Main Menu
Home | Contact Us |
Document: Done

Transfer

It's simple to move money between your accounts online.

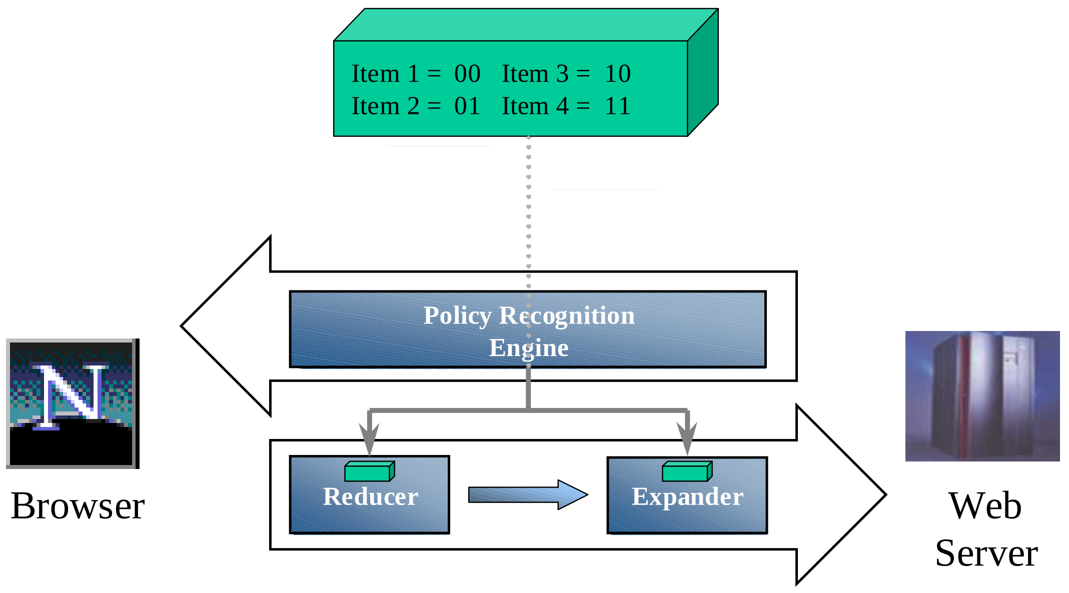
Transfer from account	Pam's Checking 9876-5437XXX
Transfer to account	John's Checking 8765-432XXX
Amount	

Set up an Automatic Transfer from one account to another.

Enforcing the dynamic policy for every user is done using *Adaptive Reduction Technology(ART)*. *Adaptive Reduction Technology* functions much the same way water is purified through distillation. The contaminated water is placed in one tank and is boiled. The steam is then transferred to a second tank in which it is turned back into water. In the process, all impurities are removed from the water, leaving it fresh and pure. Similarly, *ART* uses a *reducer* to translate any request sent from the browser into a *simple and secured language*. This secured language representation is then used to rebuild a request by an *expander*. *Reducing* the request to its simplest form and then expanding it prevents any illegal information from being passed to the application. This language is context dependent and is dynamically adapted to the current state of the application, based on the security policy set by the *Policy Recognition Engine*.

To make this point, let's imagine there's an application that sells four items: a desk, a chair, a pen, and a pencil. The application generates a page that contains four links to the four items. On its way from the web server to the browser, the page is captured by the *policy recognition engine* and analyzed. The *Policy Recognition Engine* recognizes the four links and generates a *simple and secured language* to represent the four potential links inside *AppShield*. The *simple and secured language* in this example is a two-bit code (00 for a desk, 01 for a chair, 10 for a pen and 11 for a pencil). When the user clicks the desk link, a URL is sent back to the site. The URL is captured by the *reducer*, which in turn translates the URL to 00 and sends this information to the *expander*. The *expander* then translates 00 back to the desk URL and forwards the request to the web server.

Example of Adaptive Reduction Technology in action



In case of a hacking attempt, the reduction phase of ART will fail. Instead of relaying the illegal request, AppShield invokes a customizable error CGI with information about the origin of the attack and its type. In response, that CGI generates an error page that is sent to the hacker. AppShield also invokes a Timeout CGI in case a request is sent after a session is timed-out.

Conclusion

AppShield not only provides application security, it also improves the process in the e-Business application cycle. Errors created throughout the development, testing and deployment stages will not cause security breaches within the web site. Nor would any security holes in 3rd party or public domain applications. This change in the security environment enables the organization development process to shift away from security and to its true focus: adding greater functionality to the web site. The result is a more secure web site built more quickly and offering a better overall customer experience.

More Information

1. Sanctum Inc., the WebApplication Security company.
2. The World Wide Web Security FAQ by Lincoln D. Stein
3. NT Web Technology Vulnerabilities from Phrack Magazine
4. Perl CGI problems from Phrack Magazine
5. Writing secure CGI scripts for WWW servers
6. The Unofficial Web Hack FAQ by Simple Nomad