

Common Application Security Vulnerabilities

DAS EISPD – Enterprise Security Office

April 8, 2008

An authoritative source for Web application security vulnerability information is the Open Web Application Security Project (OWASP)¹. This paper addresses several of the categories described in their “Top Ten Application Security Vulnerabilities for 2007”² and outlines how each vulnerability works and how it can be prevented. Focus is specific to vulnerabilities that the Enterprise Security Office (ESO) has direct experience with and feels are most common in the State environment. Where possible, exploit examples are given and discussed. This document is targeted to a technical state audience creating PCI-compliant e-commerce applications.

Major application vulnerabilities discussed here include:

- **SQL Injection** (a subset of OWASP's “Injection Flaws” category)
- **Cross-Site Scripting**
- **Information Leakage and Improper Error Handling**
- **Improper Cryptography and Insecure Communications** (a combination of OWASP's “Insecure Cryptographic Storage” and “Insecure Communications” categories).
- **Failure to Restrict URL Access**

Although we're only addressing six elements of it with this paper, readers are encouraged to read and understand all of the OWASP Top Ten list. It is important to protect against all ten of the major vulnerability areas, not just the ones highlighted here.

SQL Injection

Perhaps the most common and dangerous vulnerability that the ESO has encountered in application security incidents has been SQL Injection vulnerabilities. This is a subset of OWASP's “Injection Flaws” category³. Although all of the incidents of SQL Injection that the ESO has responded to have been in .ASP code running with a MSSQL backend database, all programming languages and database backends are vulnerable. As noted by OWASP, injection flaws are not limited to SQL but can effect LDAP, XML, HTML (see “Cross-site scripting,” below), underlying operating systems or any other system that gets user-data passed to it. This technique can be used to merely alter application data or can cause a complete compromise of the hosting database or the database server. Note that this attack effects the server that hosts the backend database – in a two-tier architecture where the application server is exposed in a network DMZ but the database server is positioned in a more secure network zone, this attack may compromise the protected database server rather than the application server.

How it works: Malicious user-supplied data containing database commands are passed by the application directly to the backend database. The backend database processes the

1 The Open Web Application Security Project - <http://www.owasp.org>

2 The OWASP Top 10 2007 page lists the ten most serious web application vulnerabilities as compiled in 2007 - http://www.owasp.org/index.php/Top_10_2007

3 OWASP A2 – Injection Flaws - http://www.owasp.org/index.php/Top_10_2007-A2

malicious user-supplied database commands instead of, or in addition to, the application commands. The database commands are processed with the permissions of the database user that the application is running under. Malicious data may be contained in form variables, cookie data, GET data or even data stored within the application database; any user-submitted data that is passed by the application to the backend database for processing with insufficient validation can be exploited to include a SQL injection attack. This method is commonly used to inject redirection code into application data fields that, when displayed by the application, redirects unsuspecting customers to malicious Web sites that infect their PCs with malware. Another common attack is to leverage database stored procedures to compromise the database server itself, sometimes granting the attacker access to the most protected zones within the network architecture.

Example Exploit Code and Discussion: The following snippet of exploit code was captured during an application compromise incident the ESO worked on. This was one part of a long series of SQL injection traffic that the attacker used to probe a database server and completely compromise it.

```
GET/pub.asp?op=0&id=41;CREATE%20TABLE%20[X_8099]([id]%20int%20NOT%20NULL%20IDENTITY%20(1,1),%20[ResultTxt]%20nvarchar(4000)%20NULL);insert%20into%20[X_8099](ResultTxt)%20exec%20master.dbo.xp_cmdshell%20'net%20user';insert%20into%20[X_8099]%20values%20('g_over');exec%20master.dbo.sp_dropextendedproc%20'xp_cmdshell'
```

The SQL injection begins with the semi-colon after “GET /pub.asp?op=0&id=41”. The attacker submits a valid GET request but terminates it with a semi-colon which, since the request is passed directly to the SQL backend database by the application code, the SQL database interprets as the end of the SQL statement. The rest of the injected code is then acted on as a new SQL statement. In this particular snippet, the injected SQL statements create a temporary table and use the MSSQL stored procedure “xp_cmdshell” to execute the Windows operating system command “net user.” The output from this command is inserted into the temporary table where subsequent SQL injection commands can read and display it to the attacker.

Defenses: The number one and most effective defense against injection attacks is *Input Validation*. All data that the end-user can possibly influence should be validated before being accepted or stored. Invalid data should be rejected rather than attempting to sanitize it, and validation should take the form of “accept known good” data rather than attempting to screen out “known bad” data since new attack methods are constantly being invented.

A secondary defense is to limit the database privileges that the application runs under to the fewest necessary to perform its function. This acts to limit damage if a compromise occurs. In the example exploit code above, disallowing the “create table” privilege would have prevented the exploit from working.

A third level of defense is to harden the backend database and restrict access to powerful stored procedures, again attempting to limit damage if a compromise occurs⁴. In the example above, if the xp_cmdshell stored procedure had been properly restricted the attacker would not have been able to compromise the SQL server platform.

⁴ For MSSQL Servers, consult the “SQL Security Checklist” (<http://www.sqlsecurity.com/FAQs/SQLSecurityChecklist/tabid/57/Default.aspx>) for a database hardening guideline.

For a more complete list of defenses and some language-specific suggestions, please refer to OWASP's Injection Flaws page at http://www.owasp.org/index.php/Top_10_2007-A2.

Cross-site scripting

OWASP calls cross-site scripting “the most prevalent and pernicious Web application security issue.”⁵ All Web application platforms are vulnerable to it. It can be very difficult to prevent and is made worse by Web browser flaws. It is a quickly-evolving area with new, more complex exploits being constantly developed.

The first thing to understand about cross-site scripting (XSS) is that it is not targeted toward the application platform. XSS attacks are targeted at users of an application by “reflecting” their attack using the vulnerable application. The goal of XSS attacks is to execute malicious scripts using the browser of the victim to compromise their workstation or attack other workstations or applications. This is different from most attacks against application vulnerabilities that have the goal of compromising the application itself.

The second key point to understand about XSS is that *any and all* sources of data that are displayed to end users must be filtered and encoded. It's not enough to screen data that's accepted directly via the application; any data coming from other sources may also contain XSS attack code – XSS attacks are frequently introduced at one site and displayed at others, e.g. through imported Web content or trusted business partner data.

A third point about XSS is that it is not restricted to attacks against Web browsers. Other desktop applications such as Microsoft Outlook, Adobe Reader, Adobe Flashplayer or any other application capable of executing scripts are also being targeted by these attacks.

How it works: The attacker must find some way to inject malicious HTML code into application data such that the code is redisplayed to other users. The HTML code is usually javascript but may be any scripting language supported by the end-user browser; ActiveX is another common example. The script injection may be performed directly, for example by introducing HTML into a form that will be stored and redisplayed by one application, or it may be performed indirectly by introducing malicious HTML into data that will be transferred between applications before being redisplayed. Injected script may contain the attack code itself or it may manipulate the site's own javascript code to perform the attack against the victim.

Example XSS code and exploit discussion: A simple but common attack scenario involves an attacker posting information to a site where content will be displayed to other users, for example a public discussion forum. That posted information contains javascript code that will execute on the victim's browser like the following:

```
';alert(String.fromCharCode(88,83,83))/\';alert(String.fromCharCode(88,83,83))//";alert(String.fromCharCode(88,83,83))/\';alert(String.fromCharCode(88,83,83))//--></SCRIPT>">'><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
```

except it is unlikely to generate a friendly pop-up box saying “XSS” like the above example does. A typical XSS payload will cause the browser to fetch content from another site that the

5 OWASP 2007 Top 10 – Cross-site Scripting: http://www.owasp.org/index.php/Top_10_2007-A1

attacker “owns.” The content fetched from the “owned” site commonly will attempt to exploit weaknesses in the browser or other desktop software to compromise the user’s workstation.

Defenses: For a complete discussion of defending against XSS, including some language-specific solutions, please refer to OWASP Top 10 2007-Cross Site Scripting.⁶ Here are some key points.

The first line of defense against cross-site scripting, as with SQL Injection, is *Input Validation*. The importance of screening input data and allowing only known-good data cannot be overemphasized. Because attackers are constantly evolving their attack code to slip past input filtering, injection mechanisms and encoding schemes have become very sophisticated. Example code showing hundreds of XSS variations designed for filter evasion can be found on the XSS Cheat Sheet.⁷

Additionally, specify output encoding (e.g. ISO 8859-1 or UTF 8) for each page that is displayed and encode all output to that specification before displaying. If output data contains malicious code but it becomes encoded, the browser will *display* it rather than *executing* it.

Although outside of the scope of this paper and of the application development sphere of influence, good workstation security practices are critical to protection against cross-site scripting attacks. First, disable or severely restrict javascript in your browser setting. This will reduce functionality of many Web sites but it prevents most browser-based cross-site scripting. Re-enable javascript only as necessary for site functionality. ActiveX should generally be completely disabled or strongly restricted. Keeping workstation software, including third-party applications, current on security patches will help minimize the impact of cross-site scripting attacks. Using good anti-virus software and keeping it current may also help warn of cross-site scripting attacks or prevent download of malicious payloads.

Information Leakage and Improper Error Handling

When an application reveals information about how it works this is known as “information leakage”. Information leakage is useful to attackers because it helps them shape their attack accordingly. Information leakage presented in two sections: General Information Leakage and Improper Error Handling.

General Information Leakage and Defenses Against It: Generally, information leakage can be variations, sometimes subtle, in what the application does under different circumstances. For example, if a valid query takes a few seconds to return but an invalid query returns instantly, this can help an attacker understand the difference between the two. A common leakage of information with security implications that the ESO has seen is for an application to return different information when a user tries to log in with an invalid logon ID versus with the wrong password. Although it's useful for a user to know that they've got the wrong password instead of having forgotten their logon ID, it's also useful to an attacker to know when they have a valid logon ID but not (yet) the right password.

Although OWASP has technical suggestions for guarding against general information leakage⁸, one valuable practice not mentioned is to “Think Like a Hacker.” This means

6 OWASP Top 10 2007-Cross Site Scripting: http://www.owasp.org/index.php/Top_10_2007-A1

7 XSS (Cross Site Scripting) Cheat Sheet: <http://ha.ckers.org/xss.html>

8 OWASP Top 10 2007-Information Leakage and Improper Error Handling:

evaluating an application from the frame of mind of someone trying to compromise it. All applications reveal something about how they function but sometimes the only way to determine if there are security implications to that leakage is by asking the question “how could this behavior be leveraged to gain more information or attack the application?” That, and the counter-question of how the behavior could be changed to prevent the information leakage, are key elements to designing a secure application.

Improper Error Handling, Dangers and Defenses: Improper error handling is a less subtle form of information leakage. A common application coding mistake is to allow detailed application-internal error messages to be displayed to the end user. These error messages, although useful for debugging application problems, can be extremely useful for an attacker because they reveal the application-internal details that allow an attacker to be successful. These error pages can also be used by attackers to locate vulnerable applications by searching for specific error strings on Internet search engines.

The following example illustrates two things: first, the types of information that improper error handling can reveal and how it's useful to attackers and secondly, how displaying these types of pages can draw attackers to you. This example and many more were found in a few minutes using Google and searching for generic error strings – this is a common way for attackers to find targets on the Web. Site-specific details have been changed in this example.

```
You have an error in your SQL syntax near 'and tabgroup.MMMMM= and
tabcontent.Number='1000' and tabgroup.Number=tabco' at line 1
Warning: mysql_num_rows(): supplied argument is not a valid MySQL result resource
in /customers/hostname/hosting/1234/customername/classes/class.Path.inc on line 31
```

Here we discover that the Web site is using Java (“classes”) and MySQL (“mysql_num_rows”), probably on a Unix or Linux platform (use of “/” path separators, no Windows drive letter). We see that this Web site is hosted on a shared services platform (pathnames by customer) and the internal pathname to some Java classes is revealed. Additionally, specific database table information is revealed: table names “tabgroup” and “tabcontent” with column names “MMMMM” and “Number” with valid values for them.

Although this single example illustrates specific hosting, application coding and database technologies, all application platforms are subject to this vulnerability. The data revealed in this example could be enough to draw an attacker to the site and to give them the information they need to compromise the application. Not only do error messages like these give an attacker a starting point, an application that returns this level of detail makes it easy for an attacker to craft their exploit.

Defense against this vulnerability relies on ensuring that application errors and exceptions are captured and handled by the application without displaying the internal details to the end user. Detailed practices for doing this are described by OWASP⁹. If display of error message details is necessary during the application development process, it is critical that access to the development application server be restricted to the development team and that correct error handling be implemented and fully debugged before development code is promoted to the production environment.

http://www.owasp.org/index.php/Top_10_2007-A6

9 OWASP Top 10 2007-Information Leakage and Improper Error Handling:

http://www.owasp.org/index.php/Top_10_2007-A6

Improper Cryptography and Insecure Communications

Done correctly, cryptography and encryption play key roles in protecting applications that handle sensitive information such as personally-identifiable information (PII) or credit card data. Properly encrypted data communications help maintain the confidentiality of data and ensure that applications are compliant with Payment Card Industry (PCI) standards. Encryption of sensitive data stored on a server or in a database, using well-implemented cryptography, can protect information from improper access by employees or in the case of a server compromise.

There are several ways that encryption can be incorrectly implemented that can put sensitive data at risk. One mistake that the ESO has seen several times is that data communications supposedly encrypted are passed inadvertently via a non-encrypted channel. This has usually been due to server misconfiguration where the non-encrypted channel was not shut down once SSL was configured, thus allowing users to mistakenly use the insecure channel. Another frequent problem is, although communications with end-users have been encrypted via SSL, inter-server communications take place using non-encrypted channels. It is important that any communication channel that passes authentication or other sensitive data be encrypted. Under the PCI Data Security Standard this becomes mandatory in 2008 for any channel where cardholder data is passed. For more details on securing communications channels refer to OWASP¹⁰.

When encrypting data at rest, whether in a database or on a server or backup media, there are several ways in which improper implementation of cryptography can put the data at risk.

A common problem that doesn't have a good solution is improper key management. In order to provide automated services, it's necessary for encryption keys to be available to automated server processes. This usually means storing some form of the key to decrypt data on the same server as the data itself. Care must be taken to store these keys as securely as possible. Do not hard-code them into scripts, secure permissions to them as much as possible, and use care when transporting them.

Another problem to avoid is to use high-grade, trusted encryption rather than low-grade, old or custom encryption. Encryption algorithms are continually being downgraded in strength as researchers find weaknesses in them and general computing power available to crack them increases. It is important to pick strong encryption to increase the lifespan that it will be useful. Under no circumstances attempt to design a custom encryption algorithm – cryptography is a highly-specialized discipline and amateur cryptography has been the downfall of many an application.

One important note – if an application can avoid storing or transmitting sensitive data in the first place it's going to be a much more secure application than any amount of encryption will make it. When designing an application, minimize the amount of sensitive data that is stored or transmitted if at all possible.

Refer to OWASP's Insecure Cryptographic Storage page¹¹ for additional guidelines on using cryptography and storage.

10 OWASP Top 10 2007 – Insecure Communications http://www.owasp.org/index.php/Top_10_2007-A9

11 OWASP Top 10 2007 – Insecure Cryptographic Storage http://www.owasp.org/index.php/Top_10_2007-A8

Failure to Restrict URL Access

This is a general category that covers a variety of application security mistakes. The security flaw at the heart of this vulnerability is “security through obscurity,” the mistaken assumption that if something is hidden, it's safe. The failure to further restrict access to “hidden” resources allows lucky or knowledgeable attackers to find and exploit them. The ESO has seen many different aspects of this vulnerability.

A common example is the inclusion of application or application server configuration files inside the “web root” directory where they can be accessed via the web server. We've seen this happen many times, not only with custom applications and web sites but also many commercial off-the-shelf applications. The problem with this is that, even though the documents aren't linked to from anywhere, once the path to the files becomes known they can be found on any implementation. In one case using a commonly-deployed application, once the vulnerability had been discovered in an agency implementation, implementations across the country were seen to be vulnerable to the same thing by browsing to the “hidden” configuration file location. Application and server configuration should always be stored outside of the “web root” area that can be served by the webserver. If this is not possible (i.e. with Apache .htaccess files) then access to those files must be specifically restricted.

An important variation of this vulnerability is presented in OWASP's Top Ten Vulnerabilities¹²: code that evaluates privileges on the client but fails to confirm them on the server. The ESO has seen examples of this vulnerability as well; web applications that perform some evaluation or audit on input values using client-side javascript but do not audit those values when they're presented to the server. This vulnerability allows an attacker to control those aspects of the access control or audit process, they simply need to be able to read the page source code and create their own versions of it. Although it may be appropriate to audit input values on the client side to provide immediate feedback, all submitted input must be subject to the same audits on the server side as well.

OWASP presents detailed information on protecting applications from this vulnerability¹³, but a good general protection is to never trust that hiding data will keep it secure, and never secure data solely using client-side protections.

Conclusion

In this paper we've touched on six of the OWASP's Top Ten Application Security vulnerabilities. Considerably more could be written about each of these, and it's important to understand all ten of these basic classes of vulnerability and how to prevent them. The OWASP Web site is full of information about specific vulnerabilities and how to avoid them, including coding-language-specific guidelines.

In addition to OWASP, there are a number of more formal education venues available to developers of secure applications. The ESO has compiled a list of some of these in the “Educational Resources” section, following.

12 OWASP Top 10 2007 – Failure to Restrict URL Access http://www.owasp.org/index.php/Top_10_2007-A10

13 OWASP Top 10 2007 – Failure to Restrict URL Access http://www.owasp.org/index.php/Top_10_2007-A10#Protection

Designing secure Web-based applications can be a complex task but it is a necessary one. Because attackers are able to steal large amounts of money from Web-based applications, they're developing increasingly complex and skillful attack tools. The only way to protect these applications is to implement strong security principles when designing and building them.

Educational Resources:

Microsoft has some developer security training available online from their website available here: <http://msdn2.microsoft.com/en-gb/security/aa473879.aspx> .

A major provider of technical information security education is the SANS Institute. They provide a number of courses for developers, including "SEC519: Web Application Security Workshop," "SEC536: Secure Coding for PCI Compliance" and "SEC 522: Defending Web Applications," among others including courses on specific programming languages. SANS Developer education courses can be accessed here: <http://www.sans.org/training/courses.php#developer> .