Document title:	**Procedures for secure system software development and certification**
Due delivery date:	**28/08/2012**
Nature:	**Deliverable**
Project Title:	**Collaborative Information, Acquisition, Processing, Exploitation and Reporting for the prevention of organised crime**
Project acronym:	**CAPER**
Instrument:	**Large Scale Collaborative Project**
Thematic Priority:	**FP7-SECURITY-2010-1.2-1**
Grant Agreement:	**261712**

Organisation name of lead contractor for this deliverable:

Dissemination level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	X

History			
Version	First name & Name	Modifications	Date
0.1	Felipe Melero	Fist version	2012/05/14
0.2	Carlos Monreal	Review	2012/07/16
0.3	Felipe Melero	Review	2012/08/14
1.0	Carlos Monreal	Final version	2012/08/28

Validation			
	First name & Name	Organisation short name	Visa
Responsible	Felipe Melero	S21sec	Ok
WP leader	Felipe Melero	S21sec	Ok
Coordinator	Carlos Monreal	S21sec	Ok

INDEX

1 INTRODUCTION

Security is a fundamental factor that must be borne in mind throughout the entire application development process, although it is especially important for distributed applications due to the characteristics of the environment where they are executed and more so in a system such as CAPER where confidentiality is a key point.

Knowing about and understanding the possible attacks that can be made against a system under construction help in the design of ways to avoid them or make them as difficult as possible.

This guide sets out to be a starting point for the analysis, design and implementation of security controls in each and every module that makes up the CAPER system.

2 VALIDATION AND CERTIFICATION OF END SYSTEM

The recommendations made in this document are of obligatory compliance for all the development teams that build a module included in the CAPER system.

As a final stage to certify appropriate fulfillment of the security procedures described in this document, prior to finalizing the project, expert security consultants from S21sec shall audit the end system so that it can be certified that the system fulfils the suggested procedures and standards.

This validation acts as a guarantee of the optimum security level of the overall system and enables end users to start working with it with the confidence given by using a robust, secure system.

3 OBJECTIVES AND SCOPE

The objective of this document is to act as a guide for designers and developers of the CAPER software modules when designing and implementing security controls.

It sets out to provide knowledge about the conflictive sections of the applications that require special attention during design and implementation. The main errors that are usually committed when executing the design implementation are also presented, along with the most frequent vulnerabilities of this type of application and the mechanisms that can be used to avoid them.

3.1 LIMITATIONS

Although many of the concepts presented in this guide focus on the design and implementation of web applications and communications based on Web services (the examples are developed using technologies based on .NET and Java), they are general ones and can applied to any type of technology used to implement every module making up the CAPER system.

4 CRITICAL SECTIONS OF AN APPLICATION

This chapter sets out to present the sections where special attention should be paid when designing and implementing the application due to the fact that they represent aspects that are critical from a security perspective.

4.1 AUTHENTICATION

Authentication is the process by which an entity shows its identity to another.

There are numerous authentication methods but the most commonly used one consists of providing a user name and password to the system to be accessed. The remote system accepts or rejects the connection according to whether the credentials are correct or not.

Authentication systems are one of the main objectives of a system attack. A successful attack on an authentication system guarantees access to restricted resources or information. For this reason it should also be one of the points where most resources are invested to guarantee its security.

4.1.1 Robustness of passwords

The use of short passwords, words from dictionaries, user name as password, by default passwords, etc, mean a risk for the authentication mechanism of the application.

All the passwords of the application should be robust and should comply with the following characteristics:

- *Use at least three of the following sets: upper case letters, lower case letters, numbers and special characters.*
- *Have no less than 7 characters (requirement of PCI).*
- *Not use easily guessed words or ones that are related to the user and/or company (company name, name of a son/daughter, etc).*
- *Not use words that can belong to a dictionary, or names of people, cities, etc.*

To control a minimum security policy, the application should check that the passwords chosen by users are robust. The following is an example of how to do this:

```
if password.RegEx([a-z])
     and password.RegEx([A-Z])
     and password.RegEx([0-9])
     and password.RegEx({8-30})
     and password.RexEX([!"£$%^&*()])
     return true;
```

```
else
     return false;
```

Example 4.1. Validation of robust passwords

To facilitate the selection of valid passwords (for the application) when choosing or modifying a password, a robustness indicator can be implemented that indicates to the user if the password is robust while he types it out.

Web browsers that save user passwords should be avoided. This can be achieved with the “autocomplete” attribute:

```
<input type=”password” autocomplete=”off”>
```

Example 4.2. Deactivating auto-completion of passwords

It is also advisable to not initiate sessions automatically or record previous sessions using the “Remember me in this device” function.

4.1.2 Saving passwords

For a user to authenticate with an application or web service using a user identification and password, the remote application should save the credentials in, for example, a database.

To guarantee confidentiality despite the likelihood of unauthorized access to the credential storage resource, the passwords should be saved using a *Hash* algorithm only: MD5, SHA-0, SHA-1, SHA-256, etc.

This type of function is irreversible, and so theoretically it is impossible to recover the password from its *Hash* function or summary. However, weaknesses are known in some of these algorithms that enable collisions to be found in a fixed computation time, although this is extremely high. Besides, all *Hash* algorithms are sensitive to brute force attacks in rainbow tables.

For all the above reasons, rejecting the use of algorithms such as MD5 or SHA-0 is recommended; more robust versions should be used instead, such as SHA-256. This algorithm makes it impossible to obtain collisions and up to a certain point makes rainbow table attacks difficult thanks to the large quantity of physical space required to store enough passwords so as to make an attack of this kind a success.

To carry out authentication, the applications should calculate the *Hash* function provided by the user and compare it with the one in the data base. If both summaries are the same, the password provided is correct.

```
import java.security.MessageDigest;
public byte[] getHash(String password)
  throws NoSuchAlgorithmException
{
     MessageDigest digest = MessageDigest.getInstance("SHA-1");
     digest.reset();
     byte[] input = digest.digest(password.getBytes("UTF-8"));
}
```

Example 4.3.Calculation of a Hash function (in Java)

The *hashes* or summaries of the saved passwords are unique to each one. This means that if two users utilize the same password, their summaries are also identical. To avoid providing any type of information about passwords and to make the previously mentioned attacks more difficult, the use of a salt is recommended.

A salt is a random combination of bytes (but always of the same length) that are added to the password before calculating its summary. This way, two identical passwords will have a different summary by having different salts.

Besides, if the salt is made up of any type of character, including unprintable characters, rainbow table based brute force attacks become extremely complex operations.

The salt of each password should be saved alongside it, to enable it to be used to calculate the summary of the password provided by the user that is trying to authenticate into the application.

```
import java.security.MessageDigest;
public byte[] getHash(String password, byte[] salt)
  throws NoSuchAlgorithmException
{
     MessageDigest digest = MessageDigest.getInstance("SHA-256");
     digest.reset();
     digest.update(salt);
     return digest.digest(password.getBytes("UTF-8"));
}
```

Example 4.4. Calculation of a Hash function using salt.

4.1.3 Associated vulnerabilities

The most frequent vulnerabilities related to authentication mechanisms are:

* **SQL code injection**

  The absence of validation, or an incorrect validation can lead to an SQL code injection to evade the authentication mechanism.

* **Cross Site Scripting (XSS)**

  The possibility of carrying out a code injection attack on the authentication page facilitates theft of credentials, phishing attacks, and session hijacking.

* **Obtaining sensitive information**

  Using error messages such as “User does not exist” and “Incorrect password” make it possible to obtain a list of valid system users via a brute force attack.

  By trying a large number of possible users, the system users are the ones where the error message will be: “Incorrect password”.

* **Weak password policy**

  The existence of users in the application that use weak passwords means a risk to the integrity and confidentiality of the system. For this reason applications must ensure that all users, especially those with more privileges, use robust passwords that are difficult to guess.

4.2 AUTHORIZATION

Authorization means the management of the privileges of each user that determine the functions that they can use in the application. Therefore the authorization affects a large number of tiers of the application that go from simple authorization for a certain function to access to the data base.

In every case it is necessary to use the principle of minimum privilege to guarantee that users cannot carry out actions beyond the ones that are permitted.

Authorization is especially important in multi-user environments where task segregation exists. For example, it is common for there to be administrative users who have more privileges than normal application users (vertical authorization). There may also be cases where a user must not have access to the data of another user (horizontal authorization).

Detecting insufficient authorization controls in an application is difficult due to the fact that authorization is an area that has to be made to measure for each application. The areas where errors normally arise that can affect authorization are areas with application logic:

```
if user.equals ("NormalUser") {
      grantUser (Normal_Permissions);
} else {
      grantUser ("Super_Persmissions);
}
```

Example 4.5. Control of a user's privileges

A first mechanism to prevent users from accessing functionalities that should not be enabled for them is to not show these utilities in the user interface (GUI). This way it is difficult for a user with low privileges to know about functionalities to which he does not have access.

This "concealment" of information should not be carried using hidden fields since the user can always access the source code of the web page. Besides, if the web application permits another crawler application to scan the web in search of links, they can appear in the report that the tool generates for the user.

Neither is it advisable to disable the use of button on the user side, since they might not impede the running of associated functions.

```
document.form.adminfunction.disabled=true;
```

```
<form action="./doAdminFunction.asp">
```

Example 4.6. Disabling buttons on the client side

In the previous example, it would be enough for the user to save the web page in his computer, modify the code to enable the button and enter the complete route of the Action form to execute the function for which he has no privileges.

The concealment of information should not be the only mechanism to authorize an application, as it leaves all the responsibility for this on the client side, and is therefore easily annulled (e.g. by knowing the URL of a function).

The correct way to implement it is that in all the requests made by users, their privileges are checked to determine if the requested action can be executed or not.

The following example shows how a user is checked to see if he has privileges to carry out a certain action he has requested. If he has them, the action is executed and if not, he is notified and the session is invalidated:

```
String action = request.getParameter ("action")

if ( action == "doStuff" ) {
      boolean permit = session.authTable.isAuthorised (action);
}

if (permit) {
      doStuff ();
} else {
      throw new InvalidRequestException ("Unauthorised request");
      session.invalidate ();
}
```

Example 4.7. Control of authorization

4.3 SESSION MANAGEMENT

Session management creates, renews and destroys user sessions during the execution of an application.

Correct session management should guarantee that:

* The identifier of the session should be robust and cryptographically secure, making prediction impossible. It is recommended that the generation of session identifiers is left to the running environments (virtual machines or frameworks).

* A new session identifier should be created every time a user is authenticated in the application, and his validity should last for as long as the session does, being destroyed with it after request by the user with the “End session” function or after a period of inactivity.

* The session identifiers should be sent to the user via secure means. The use of identifiers in URLs should be rejected as these are usually registered in server log systems. Instead of this, they should be exchanged with the user via secure cookies to prevent them from being sent via unencrypted channels.

```
cookie.setSecure() (Java)
cookie.secure = secure; (.NET)
```

Example 4.8. Use of secure cookies

* The application should check that the session is valid along with the privileges of the user associated with it (authorization) before taking any action.

* The application should provide a function to end the user session that invalidates the session to prevent it from being used in later situations.

A valid mechanism to prevent the theft of sessions is to associate each user with the session identifier assigned to him after authentication and the IP address from which he accessed the application for the first time. This way, even if it is possible to capture the session identifier, a second user who wants to supplant the identity of the first one cannot do so because he is obliged to access the application from the same IP address.

For total control over the session and to impede session theft as well as CSRF attacks, the use of nonces is recommended. Nonces are random numbers that are used once, which are sent to the user and then expected when he returns it in the next request. If the received nonce is different from the one sent, then something has happened and the session should automatically end.

The nonce can consist of a summary or hash of a number returned from a PRNG (pseudo-random number generator) that is saved in the session of each user:

```
byte[] nonce = new byte[16];
Random rand = java.security.SecureRandom .SecureRandom.getInstance ("SHA1PRNG");
rand.getBytes (nonce);
```

Example 4.9. Implementation of a nonce

The classes that each language implements to make this process a simpler one can also be used.

```
Microsoft.Web.Services3.Security.nonce
```

Example 4.10. Example of implemented nonce.

4.3.1 Associated vulnerabilities

The most common vulnerabilities that can mean a risk to session management are:

* **Cross Site Scripting (XSS):**

  Session theft is possible by means of code injection attacks.

  To prevent a script code from accessing cookies, the flag HTTP Only can be used, although this measure is only valid for the Internet Explorer browser from version 6 onwards.

```
cookie.HttpOnly = true (C#)
```

Example 4.11. Implementation of HTTPOnly

* **Cross Site Request Forgery (CSRF):**

  With this type of attack a legitimate user of the application can be forced to execute functions without his knowledge. For example, an attack of this type could be used to change the password of the user to another one that is known by the attacker.

* **Session fixation:**

  Session fixation occurs when an application allows a user to be authenticated without eliminating the identifier of the previous session. This enables an attacker to force a legitimate user to authenticate in the application by using the known session identifier so that, after authentication, the attacker has access to the application with the privileges of the authenticated user.

4.4 ENTRY VALIDATION

Many of the attacks explained in this guide are based on the lack or incorrect validation of user entry parameters. Some attacks that can be prevented using correct validation of user entry are:

- SQL code injection
- HQL code injection
- LDAP code injection
- XPath injection
- Cross Site Scripting
- Injection of commands from operating system
- Execution of external applications
- Etc.

In all these vulnerabilities it is often necessary to enter some special characters such as single or double quotation marks (‘’‘, ‘“’), mathematical symbols (‘<’, ‘>’, ‘=’, ‘*’, ‘/’, ‘%’, etc.), punctuation marks (‘;’, ‘.’, ‘_’, etc.), etc.

It is also common to include words reserved from the language that is to be injected: LIKE, UNION, GROUP BY, HAVING, script, href, exec, etc.

To prevent risks of injection, it is recommended that **all** the entries of the user are validated in as restrictive a manner as possible. To do this:

1. All entries should be generalized and expressed in the same codification, for example ASCII 256. If not, some codifications can confuse the validation filters and permit undesired code injection.

2. White lists should be used to validate that the characters entered by the user are the expected ones.

   In situations in which only a finite (and small) number of entries is permitted, it is possible to make white lists including all the possible values. In other situations regular expressions should used for validation.

   Some examples of regular expressions are:

- Only alphabetic characters (the empty chain is not accepted): **^[a-zA-Z]+$**
- Only alphabetic characters (the empty chain is accepted): **^[a-zA-Z]*$**
- Only numeric characters (the empty chain is not accepted): **^[0-9]+$**
- Only numeric characters (the empty chain is accepted): **^[0-9]*$**
- One of a finite set of characters: **[abcde]**
- None of a finite set of characters: **[^abcde]**

More examples of regular expressions can be seen in Appendix A.

Checks can also be made on the length of the entered chains, no allowing more characters than the ones expected, for example, postal codes, dates, telephone numbers, national identity card numbers, passport numbers, etc.

Fields that are not a chain type (whole, Boolean, dates, etc) should be converted to their type by special functions.

```
Int32.TryParse(value, out number); (C#)
int number = Integer.parseInt (value); (Java)
```

Example 4.12. Conversion to native data types

Complex constructions for which preparing a regular expression is a costly process, should be validated by parts to guarantee the validity of the data provided by the user.

3. Finally filtering of reserved words from the language to be prevented (black list validation) is also possible: SQL for access to data bases, JavaScript and HTML for values that are returned to the user in a web page, etc.

It should be highlighted that the use of validations by black list does not always ensure that it is not going to be injected. For the validation process to be as effective as possible, following the previous three steps is recommended.

Another aspect to bear in mind when implementing validation is that it should always take place in the server. Although validation mechanisms can be implemented in the client (JavaScript) to free the server from the load, they should be repeated on the server side to confirm that the entry data is valid.

To prevent Cross Site Scripting attacks, it is recommended that all the outputs to be shown in the view are codified using HTML codification.

```
HttpUtility.HtmlEncode (string); (C#)
org.apache.commons.lang.StringEscapeUtils.escapeHtml(String str); (Java)
```

Example 4.13. Output codification

Generally speaking, before issuing any output that is processed by an interpreter (web browser, web service, data base, etc), it should be codified to avoid possible injections in the remote interpreter.

4.5 cryptographic controls

Cryptography provides mechanisms that enable data confidentiality to be maintained (by means of encryption), it also ensures data integrity (by means of summary functions), impedes non-repudiation (by means of digital signatures) and authenticates entities (by means of digital certificates).

However, for the above mentioned processes to be effective, the selection of the adequate algorithm and key is a fundamental point within a system design process.

As a general rule, it is recommended that all communications are encrypted, especially the ones that involve an exchange of sensitive information such as user credentials, financial information, information of a personal nature, etc.

On the other hand, it is recommended to use standard implementations provided by programming languages instead implementing versions of the algorithms, since these can be linked to failures of implementation that diminish or cancel the cryptographic properties of the algorithms.

4.5.1 Symmetric encryption algorithms

In symmetric cryptography, it is necessary for the parties involved in the communication process to share a secret (code) that is used to bring about the encryption and decryption processes of the message.

Symmetric algorithms are usually much faster than asymmetric key algorithms, and so are used in communications where it is necessary to exchange large amounts of information or simply to encrypt data to which user access is not desired. In the latter case, the key is only stored in the server, and so it is impossible to decipher the message even when access is gained.

To resolve the problems involved in letting both parties know the key, there are many algorithms to exchange keys via a network (insecure) that ensure that the key that is finally used does not travel at any time through the resource and that it is impossible to be calculated from the information that the parties exchange.

These algorithms, on the other hand, do not work for communications where the intention is to send messages from one to many, as this would mean sharing they key with each and every recipient.

When choosing a symmetric key algorithm, the use of DES should be rejected, which despite being cryptographically secure, has a key length (56 bits) that makes it sensitive to brute force attacks.

The use of Triple-DES (3DES) is recommended instead, which consists of the concatenation of three DES ciphers using a 128 bit key. As can be imagined, this algorithm is slower than DES but much more secure. Another faster and safer alternative is the Rijndael or AES algorithm, which is the current encryption standard.

The length of the key in this type of algorithm should not be less than 128 bits. However, with a view to future computing capacities that might enable brute force attacks for these lengths, it is recommended to use keys of 168 or 256 bits.

4.5.2 Asymmetric encryption algorithms

This type of algorithm, unlike symmetric key algorithms, uses a different key to encrypt and decrypt a message. This way there are two keys known as public and private dues to the fact that one should be known by anyone with whom information exchange is necessary (public) and the other that should only be known by the proprietor (private).

Asymmetric cryptography enables information from one to many to be exchanged; it also permits integrity of communicated information and avoids non-repudiation. These two final uses are achieved by combining public key cryptography with summary functions.

The main asymmetric key algorithm used nowadays is RSA. This algorithm is used in a large number of applications, of which PGP and SSL are notable examples.

This type of cryptography is also used in preparing digital certificates that are used to authenticate entities with greater security than when exchanging credentials (user name/password).

The digital certificates are issued by a Certifying Authority (CA). When entity A obtains a unique certificate, it is sent to B, which is the entity where authentication is desired. If B trusts in the CA that the A certificate issued, the authentication is satisfactorily completed.

For this reason it is important that when validating a digital certificate, it is checked that the CA that issued it is trustworthy. If this is not the case, the certificate must be rejected and authentication should not be completed.

The length of the selected key for this type of ciphering should not be less than 1024 bits. However, for the same reasons as those mentioned for symmetric algorithms, keys of 1536 and 2048 bits are recommended.

4.5.3 Hash or summary functions

This type of function prepares a summary or hash of fixed length for a message.

The properties of this type of function ensures that the summary is easy to calculate, that the process is irreversible (the original message cannot be obtained from the summary) and the summaries obtained for similar messages are very different.

This type of function is ideal for saving passwords, as they avoid the possibility of recovery when accessing their summary. They are also used in message signing and non-repudiation processes

There are a large number of hash algorithms; the most well known are: MD5, SHA-0, SHA-1 and SHA-256. At present, the first two (MD5 and SHA-0) are regarded as weak because it is possible to find collisions in a finite period with sufficient computing capacity Although SHA-1 continues to be robust, it has been possible to weaken it using design-related weaknesses. For this reason SHA-256 is recommended whenever possible.

4.5.4 Secure distributed applications

There are many alternatives when constructing a distributed application. A classification depending on the type of communication that needs to be implemented might be:

* *Message passing oriented communication:: Sockets, MPI, etc.*
* *Distributed object architectures: RPC, CORBA, RMI, etc.*

- *Service oriented architectures: SOAP (used in CAPER), etc.*
- *Status transfer oriented architectures: REST, etc.*
- *Client-server architectures: Web applications (CAPER Management Application, Visual Analytics tools), etc.*

Independently of the chosen architecture, when implementing a distributed application, there are four essential factors common to all of them. **sender, receiver, channel and message**.

It is recommended that obfuscated messages are transmitted if the application to be implemented transmits data that is regarded as sensitive (credentials, bank data, personal information, etc). To implement obfuscation, cryptographic controls can be used.

There are two alternatives for implementing obfuscation:

- *Information encryption: Any of the robust algorithms mentioned in this chapter can be used to encrypt sensitive information and transmit it through the channel.*
- *Channel encryption: The channel can be encrypted so that the information transmitted through it is obfuscated. For example, SSL, which is a recommended option in the deployment of the CAPER modules, which acts in the transport tier and provides the advantages of a PKI: authenticity, confidentiality, integrity and non-repudiation.*

  In this case there are several alternatives; depending on the chosen technology it is possible that:

  - *The use of the channel is encapsulated and transparent (e.g. RMI) or by direct use of the channel (e.g. passing the message via sockets). In both cases it is usually possible to define the type of socket to be used, so that secure sockets can be used (SSL), bearing in mind all the implementation considerations of a secure PKI (authentication against the certificate, verification of the CA, expiry, etc).*
  - *Messages are sent via a protocol that supports the use of an encrypted channel. In these cases the use of a channel is planned that obfuscates the information, and the protocol transparently implements the encryption on the transport tier (e.g. HTTPS).*
  - *If there is no control over any of the ends (sender, receiver or both) when encrypting communications, mechanisms can be used at system level below the application tier (e.g. IPSEC).*

Other important factors are already described in other sections of this guide such as: **Authentication, Authorization, Entry Validation** (in this case the message, etc), etc.

4.5.5 Associated vulnerabilities

The vulnerabilities most commonly associated with the use of cryptography are:

* **Weak encryption:**

As can be observed throughout this section the choice of algorithm and key is an essential factor for making a cryptographic system work as expected. The incorrect selection of any of them means that the system can be compromised or not offer the expected features.

* **Lack of encryption in communications:**

Sending unencrypted information via insecure channels, such as a local network or the Internet, means that any user with access to the channel can access the transmitted information. Besides, if the information is of a sensitive nature such as credit card numbers, users' personal data, application credentials, etc. This may mean a risk for other users or the application itself.

4.6 event log

Preparing a log of the events that occur in an applications enable an error to be detected and corrected in cases of failure or intrusion, or the steps taken by an attacker to be reconstructed.

The events that should be stored include the following:

* Starts and ends of sessions. Unsatisfactory ends should also be logged to detect brute force attacks.
* Requests requiring some kind of authorization.
* Data handling: creation, modification or deletion of information.

It is also possible to have and event log for actions that cause errors in the application so errors in it can be detected and corrected.

Under no circumstances should user entries be logged that have not been correctly validated as a result of possible injection problems in the event log systems (such as a data base) or malfunctioning of the tools that interact with them.

Storage of sensitive or confidential information should also be avoided. To resolve this need internal applications identifiers can be used (user identifiers, process identifiers, etc) instead of names of users, processes, etc, that are to be stored. It is also possible to use encryption for information considered to be confidential and requiring logging.

The log files can be stored in encrypted form and in a location that cannot be externally accessed.

4.7 memory management

A buffer is an amount of contiguous memory that is used to store information. A buffer overflow happens when the data written in a buffer corrupts the information from the memory addresses adjacent to the ones reserved for the buffer. This is usually due to a lack of validation of the input data or poor management of the memory reserves.

In modern programming languages, such as C# or Java with no control over the memory assigned to each variable, these errors are usually caused by the use of libraries (public or private) with which vulnerabilities of this type are associated.

However, these languages also support management of the memory by the programmer. For example, in C#, low level methods of memory management can be used by means of the definition of unsafe contexts. Or even the inclusion of code in other languages is permitted (such as C or C++) in ones where this type of error takes place.

There are a large number of different types of buffer overflow, but generally speaking they are caused by:

* the use of arrays and mice:

```
char buffer[5];
char *stringaux = strdup ("This will produce a buffer overflow");
strcpy (buffer, stringaux);
```

Example 4.14. Buffer overflow 1

* formatting of chains with functions such as printf, fprintf, etc.

```
printf (“%s%s%s%s”);
```

Example 4.15. Buffer overflow 2

This example shows a search in the stack up to four chains, causing a buffer overflow.

* unsafe functions such as strcpy, strcat, sprintf, etc. (such as in the example above).

* overflow of integers

```
int val = 0x7fffffff
val = val + 1
```

Example 4.16. Buffer overflow 3

The maximum value that an integer can take in C is 0x7fffffff, which corresponds to the number 2147483647. In the second instruction, *val* would take the value 0x80000000 , which corresponds to the number -2147483647. If a vector is being run, this would cause access to indeterminate memory zones, causing a buffer overflow.

5 MOST FREQUENT VULNERABILITIES

The section above considered different aspects when designing and implementing an application. Many of the vulnerabilities that affect them were also described. This sections describes each of these vulnerabilities to as to better understand what they consist of and how to mitigate them.

The criteria followed to establish criticality of the vulnerabilities are as follows:

Risk Level	Description
High	Vulnerabilities that involve total or partial access to the server or its environment. This includes other servers or equipment, storage systems such as data bases, etc. Other high risk vulnerabilities include those that permit the operations of the application to be altered in some way.
Medium	Vulnerabilities that permit information to be obtained that is sensitive or that requires third party intervention for exploitation, such as code injections into the client.
Low	Vulnerabilities that require local access to the system or that facilitate information about it. Also includes are ones deriving from incorrect user and password policies.
Potential	Cases where it cannot be determined if there is a risk because not all the necessary information is available (for example: some of the procedures or objects used are not included).

The table below shows the most common vulnerabilities with their criticalities in order of frequency of appearance.

Injection of code in client	**Medium**

Encryption problems	**Medium**
Injection of code in server	**High**
User entry logging	**Low**
Log of sensitive information	**Low**
Weak user and password policy	**Low**
Redirect	**Low**
Abuse of functionality	**High**
Execution of external applications	**High**
Buffer Overflow	**High**

5.1 Injection of code in client

5.1.1 Description

The vulnerability of Cross Site Scripting (or client-side script injection) consists of the possibility of injecting script (JavaScript, VBScript, HTML or any technology supported by the browser) distant from the application inside a web site that is executed as another part of the application.

Code injection can take place in the following ways:

- *The script is entered in the application from an unreliable source, usually a web request. The script is entered in the dynamic content that is sent to the application without it being validated.*

When it is executed, the inserted script has unrestricted access to all the information to which the browser has access, such as session cookies or other data sent from the browser.

The main types of Cross Site Scripting are as follows:

- *Non-persistent (indirect or reflected): Requires the user to visit a web address link containing the script to be entered. When visiting the link, the script is repeated and executed inside the user's web browser. What then takes place is a modification of the values sent by the user to the application and which are used to show the next page of the application. An example of this type of vulnerability is to include the search made in the results page without HTML codification, this enables script to be injected from the client side into the dynamic page. This is the most common type of Cross Site Scripting.*

- *Persistent (direct, stored or second order): This consists of entering HTML script into the web application, which stores it in the server (in a data base, file systems, visit log, comments field, etc) and then is shown to users on a web page. An example of this type of vulnerability can be found in message forums, where users can send messages in HTML format for other users to read.*

- *DOM-based (local Cross Site Scripting): produced when part of the JavaScript code accesses a parameter of a web request, uses this information to write HTML script in its own page and this information is not codified using HTML entities. This entered script is reinterpreted by browsers such as HTML and it is here where script can be entered by the client.*

5.1.2 Example:

```
<input type="text" name="xss" value="" />
<asp:Label ID="LblTest2" runat="server" />
<asp:Button ID="BtnXSSParams" runat="server" Text="Parameter"
OnClick="XSSParameters" />

[…]

protected void XSSParameters(Object sender, EventArgs e)
{
    LblTest2.Text = (string)Request.Params["xss"];
}
```

Example 5.1. XSS in .NET

```
<%= request.getParameter ("XSS") %>
```

Example 5.2. XSS in Java

5.1.3 Consequences

Vulnerabilities of this type can have several effects, such as the modification of the appearance and content of a web page, rerouting of users to false websites or theft of authentication credentials of a legitimate user. Modification of the appearance of a web page (e.g. adding fields in a form) can be used to steal identity, these security risks are known as phishing.

5.1.4 Risk

Medium

5.1.5 Solution

Certain technologies such as Struts in Java or ValidateRequest in .NET, implement validation mechanisms for entries or escape from outlets. These measures can considerably limit the exploitability of vulnerabilities but they do not totally eliminate them. There are also vulnerabilities associated with these technologies that can be exploited

It is therefore recommended to use these mechanisms as secondary measures and

implement specific validation mechanisms for all user entries. There are two methods for this validation:

- *White lists: Consist of accepting the characters that are strictly necessary for each entry. For example, if the field to be validated is a postal code, the application should only permit chains of number of 5 in length "08910", any entry that does not coincide with this should be considered as invalid. This can be done with regular expressions.*
- *Black lists: Consist of searching for patterns of possible script injections (loose quotation marks or more - less symbols (<>)). The problem with this approach is that it is impossible to ensure that all the possible injection patterns have been included, as new forms can appear.*

The white list approach requires more effort from developers but is more effective in reducing problems of this type.

It is also recommended that before showing the response page for a request made by the user, the entire document is encoded using HTML entity encoding.

5.2 Encryption problems

5.2.1 Description

Selection of the protocol, the encryption algorithm and length of the key determine the difficulty of access to the information that is to be protected from unauthorized access.

In the case of distributed applications or others via Internet, the use of protocols that utilize encryption is recommended. Some possible solutions are the use of TCP/IP over SSL with protocols such as SSH, HTTPS, FTPS or the use of VPNs.

The selection of the encryption algorithm is especially important for storing sensitive information in places the user can get access to, such as cookies, HTTP headers, etc. In these situations it is advisable to reject the use of weak algorithms, such as DES, which can be replaced by safer solutions such as Triple-DES or AES.

The key length is also fundamental in algorithms that support different lengths such as AES or RSA. Generally speaking, for symmetric encryption algorithms passwords should be used that are over 128 bits while keys of over 1024 bits in length are recommended for asymmetric encryption algorithms.

For applications that use authentication via digital certificates, verification from the Certifying Authority (CA) and the certificate itself are recommended (expiry date, entity for which it was issued, etc).

5.2.2 Example:

```
<wsdl:service name="Service">
  <wsdl:port name="ServiceSoap" binding="tns:ServiceSoap">
    <soap:address location="http://www.foo.es/foo/service.asmx"/>
  </wsdl:port>
  <wsdl:port name="ServiceSoap12" binding="tns:ServiceSoap12">
    <soap12:address location="http://www.foo.es/foo/service.asmx"/>
  </wsdl:port>
</wsdl:service>
```

Example 5.3 Unencrypted Webservice

5.2.3 Consequences

In the case of applications that require communication via a network, the absence of encryption or the use of unsafe protocols or keys may mean that any person with access to the channel can also access the information that is being transmitted: credentials, sensitive information, etc.

The same thing happens when the information is encrypted using an unsafe algorithm or with a key that is not sufficiently robust,

In encrypted communications where digital certificates are used to authenticate the other party, incorrect validation of the certificate can permit identity theft, permitting access to restricted information and resources.

5.2.4 Risk

Medium

5.2.5 Solution

Use encryption mechanisms for all the communications by means of safe protocols or VPNs.

For communications between the analysis and orchestrator modules, use secure communications mechanisms such as WS-Security.

Select robust encryption algorithms (Triple-DES, AES, RSA, etc) and high or medium key lengths, disabling the use of low or zero key lengths.

The following link can be consulted to disable low key lengths in IIS:

http://support.microsoft.com/?id=245030

To disable the low encryption ciphers in Apache, consult the following link:http://support.microsoft.com/?id=245030

To disable low encrypted key lengths in Apache, consult the following link: http://httpd.apache.org/docs/2.0/mod/mod_ssl.html

To disable low key lengths in IBM HTTP Server it is necessary to specify the algorithms that use medium or high key lengths in the SSLCipherSpec directive: http://www-01.ibm.com/software/webservers/httpservers/doc/v1326/manual/ibm/9acdssl.htm#sslcipher

When authentication is carried out via digital certificates it is necessary to check that Certifying Authority (CA) as well as the certificate data.

5.3 Injection of code in server

5.3.1 Description

The risk of code injection in the server takes place when a user entry that is not correctly validated is used for other operations. When these inputs end up being used in sentences that are then executed in other environments, such as a web service or application, data base, etc, cases of code injection can arise.

Depending on the type of sentences the user inputs use, different types of injection can occur. When the statements are SQL it is called SQL injection, when the statements are destined for an LDAP server, LDAP injection, etc. (XPath, Host, operating system orders, etc).

5.3.2 Example:

```
<asp:TextBox ID="TxTInput" runat="server" />


[…]


string request = "SELECT * FROM users WHERE name = '" + TxTInput.Text +
"'";
```

Example 5.4. SQL code injection

5.3.3 Consequences

Code injection can mean partial or complete access to data bases, XML files, LDAP servers ,etc.

Besides, in some cases it is possible to alter the information from these data stores, detain services and even execute system orders.

In the previous example, if the user enters the following chain:

```
Anthony'; DROP TABLE users; --
```

Example 5.5. SQL injection pattern.

The final query to be sent to the data base would be:

```
SELECT * FROM users WHERE name = 'Anthony'; DROP TABLE users; -- '
```

Example 5.6. SQL injection result.

If the data base server and the driver used support sending several instructions in one single request and the user of the data base has enough privileges, the above injection would mean the loss of the data from the "users" table.

5.3.4 Risk

High

5.3.5 Solution

Validation of all the user inputs is required, especially ones executed in other environments.

Validation should be as strict as possible, for which the use of white lists that only accept expected values is recommended. The use of regular expressions can speed up validation of user inputs and also enable them to be reused between applications or fields with similar characteristics.

For some specific types of injections, such as SQL code injections, the use of collections of parameters such as SqlParameterCollection in .NET, provides an additional prevention measure that avoids code injections in the data base. This is achieved by treating the input as a literal value, so that the data base does not execute the code it might contain.

It should be remembered that the collections of parameters are not really carrying out input validations, and so when they are recovered from the data base, they can cause other client-side and server-side injections. An example of this would be persistent Cross Site Scriptings, which could be stored in the data base without causing an injection there.

5.4 User entry logging

5.4.1 Description

It is common to store information entered by the user in log files or data bases. If this information is not correctly validated, it can have a negative impact on the log systems themselves, or on the tools that interact with them.

For example, if a data base is used as a log system, a user input that has not been correctly validated may mean a risk of SQL code injection in the data base.

5.4.2 Example:

```
<asp:Label ID="LblName" runat="server" Text="Name: " />
<asp:TextBox ID="TextBoxName" runat="server" />
<asp:Label ID="LblAge" runat="server" Text="Age: " />
<asp:TextBox ID="TextBoxAge" runat="server" />

order_BD = "INSERT INTO log ("name", "age") VALUES ('" +
TextBoxName.Text + "', '" + TextBoxAge.Text + "');”
```

Example 5.7. .NET user input logging

5.4.3 Consequences

Depending on the type of log system used, server-side code injections can take place as well as other unplanned results, such as a service denial or buffer overflow.

5.4.4 Risk

Low

5.4.5 Solution

There are two possible solutions to this vulnerability:

- *Validate all the user inputs as restrictively as possible. This consists of using white lists that only accept expected values. Regular expressions can be used to determine if a value is accepted or not.*

- *Not log any information from the user.*

The second solution is recommended because on many occasions users enter sensitive information in applications (personal details, credit card numbers, etc) to which personnel in charge of revising the log system should not have access.

To store information such as the user connecting to the application, it is recommended to use internal application identifiers (user identifier) that are not descriptive and over which the user has no kind of control.

If it is necessary to store other types of information provided by the user, it is necessary to make use of the first proposed solution to ensure that the stored user inputs do not involve any risk.

5.5 Log of sensitive information

5.5.1 Description

Personal information is any data that directly or indirectly permits the physical, physiological, psychological, cultural or social identity of an individual to be determined.

Critical information for systems is regarded as any data that means a sizeable risk to system security, such as user names (even when they are summary functions), source codes of the application, data base structure, etc.

It is not recommended to store this type of information in log systems as a user that has access to the system also has access to this data.

5.5.2 Example:

```
<asp:Label ID="LblLogin" runat="server" Text="Login: " />
<asp:TextBox ID="TextBoxLogin" runat="server" />
<asp:Label ID="LblPass" runat="server" Text="Pass: " />
<asp:TextBox ID="TextBoxPass" runat="server" TextMode="Password" />
<asp:Button ID="BtnLog" runat="server" Text="Logins"
OnClick="LoginSensibleInput" />

[…]

protected void LoginSensibleInput (Object sender, EventArgs e)
{
      StreamWriter writer = new StreamWriter("C:\\testLog.txt", true);
      writer.WriteLine("Login: " + TextBoxLogin.Text + " " + "Pass: " +
TextBoxPass.Text);
      writer.Close();
}
```

Example 5.8. Log of sensitive information

5.5.3 Consequences

Generally speaking, the personnel responsible for revising the log files are not authorized to access sensitive information of the system or application users. When sensitive information is stored in these systems, any user with access to them can recover it.

5.5.4 Risk

Low

5.5.5 Solution

No information that is considered to be sensitive should be stored in the log systems. If it is necessary, encryption is recommended to ensure information confidentiality.

5.6 Weak user and password policy

5.6.1 Description

A weak user name and password policy can facilitate access to system resources by illegitimate users. A password policy is regarded as weak when one or more of the following conditions are fulfilled:

* There is no assigned password.
* The password is the same as the user name.
* By default SNMP passwords or communities are used.
* Words, names or numbers are used that are related in some way to the system or user and which can be discovered.
* The password is a word which someone looking over the user's shoulder can easily reproduce. For example: 123456.
* Words are used that can be found using specific applications: words in dictionaries, company names (especially the user's), names of famous people, own names, any of the above written backwards, any of the above with a number or character inserted at the end or beginning, any of the above in a foreign language, etc.
* The passwords have less than 8 characters.
* Only one set of characters is used from the following: upper case letters, lower case letters, numbers, and special characters.
* The password is stored in the system, either in clear text or encrypted.
* The passwords do not expire.

5.6.2 Example:

```
protected void UserPasswordPolicy()
{
      Logins = new System.Collections.Hashtable();
      Logins.Add("john", "blue");
      Logins.Add("bob", "bob");
      Logins.Add("martin", "martin");
      Logins.Add("mario", "red");
      Logins.Add("jamie", "brown");
      Logins.Add("paul", "cyan");
}
```

Example 5.9. Weak password policy

5.6.3 Consequences

A weak user and password policy makes it easy for illegitimate users to access confidential resources or information.

Likewise, if the application allows passwords to be changed, it is possible to steal user accounts by accessing an account and changing the password.

5.6.4 Risk

Medium, Low

Depending on the type of application and its functionalities, the consequences for the system can vary from simple access to the applications or private zones inside it to total control over the server (e.g. via an administration panel).

5.6.5 Solution

Mechanisms should be implemented that ensure that the passwords selected by users are robust and comply with all the characteristics that were commented on in the section on vulnerability.

The application should never store passwords in clear text or in encrypted form. Although theoretically encrypted storage impedes recovery of the passwords, the confidentiality level is based on the encryption algorithm and key that are used.

To enable authentication via passwords, the application should only store the summary function of the password (MD5, SHA-1, SHA-256, etc.). This way, if access is gained to the password summaries, they cannot be recovered from there since this type of procedure is irreversible. Authentication in these cases is carried out by calculating the summary of the password provided by the user and comparing it with the summary stored in the application.

5.7 Redirect

5.7.1 Description

A redirect can take place when the application uses a user input parameter to decide the next URL that should be returned to the user or to generate a link that the user can click on.

If the input parameter is not correctly validated, the user can manipulate it to access other parts of the application or redirect to third party servers. The latter can be used for phishing attacks.

In many cases the redirect can be caused by a vulnerability of client-side code injection (XSS) that makes it possible to modify the HREF tag of a hyperlink, inject JavaScript code to automatically redirect, etc. It can also be caused by the use of methods such as HttpServletResponse.sendRedirect in Java or in System.Web.HttpResponse.Redirect in .NET.

In any case the main cause of a redirect is due to a lack of validation of user inputs.

5.7.2 Example:

```
<asp:TextBox ID="TxTInput" runat="server" />
<asp:Button     ID="BtnRedirection"     runat="server"     Text="Redirect"
OnClick="Redirection" />


protected void Redirection(Object sender, EventArgs e)
{
      HttpResponse.Redirect (TxTInput.Text);
}
```

Example 5.10. Redirect

5.7.3 Consequences

The consequences associated with a user being able to modify the normal flow of the application consist of access to restricted zones of the application and phishing attacks.

If the redirect can be made by manipulating input parameters via the GET method, the risk is increased. A user that is deceived into using a conveniently manipulated link to bring about a redirect, can be redirected to an external URL in which, for example, his system access credentials or other sensitive information can be requested.

5.7.4 Risk

Medium, Low

Depending on if the application allows the application flow to be modified to an external web or another point of the application, the risk is Medium or Low respectively.

5.7.5 Solution

Validation should be carried out on all the user input parameters that are to be used to determine the flow of the application.

Validation should be as restrictive as possible, in other words, via a white list that only permits certain valid URLs from the current application status.

In this case, the use of regular expressions would not be as effective as the definition of a list containing the valid URLs, although validation is also possible using them.

The possibility should also be considered of modifying the application logic so that the input parameter is not directly used to generate the new URL but to decide which of a number of possible ones it might be.

5.8 Abuse of functionality

5.8.1 Description

This type of vulnerability arises when a legitimate functionality of the application is improperly used to consume or use resources, evade access control mechanisms, etc.

An example is a platform that enables e-mails to be sent with no authentication process. A user can take advantage of this functionality for mass electronic mailings from a trusted entity.

5.8.2 Example:

```
<asp:Label ID="LblSendMail" runat="server" Text="Send Mail" />
<asp:Label ID="LblTo" runat="server" Text="To:" />
<input type="text" name="to" /></td>
<asp:Label ID="LblSubject" runat="server" Text="Subject:" />
<input type="text" name="subject" /><
<input type="text" name="body" />
<asp:Button ID="BtnMail" runat="server" Text="Send"
OnClick="FunctionalityAbuse" />

[…]

protected void FunctionalityAbuse (Object sender, EventArgs e)
{
      MailMessage msg = new MailMessage();
      msg.To.Add(new MailAddress("xxxx@xxxx.com"));
      msg.From = new MailAddress(Request.Params["to"]);
      msg.Subject = Request.Params["subject"];
      msg.Body = Request.Params["body"];
      SmtpClient clientSmtp = new SmtpClient("smtp.xxxx.com", 587);
      clientSmtp.EnableSsl = true;
      clientSmtp.Credentials = new NetworkCredential("xxx@xxx.com",
"xxx");
      try
      {
```

```
            clientSmtp.Send(msg);
      }
      catch (Exception ex)
      {
            LblTest.Text = ex.Message;
      }
}
```

Example 5.11. Abuse of functionality

5.8.3 Consequences

The consequences depend solely on the application and the control mechanisms it implements.

Some examples are service denials, mass electronic mailings, access to restricted zones of the application, etc.

5.8.4 Risk

High

This depends exclusively on the functionality that might be compromised.

5.8.5 Solution

Control mechanisms should be implemented to limit the scope of the application functionalities. This scope should be specified in the design phase and it is there where the controls that enable limitation should be developed.

In most cases, the control mechanisms are limited to checking if the user wanting to carry out the action has the necessary privileges to do so.

In other cases, it possible to modify the design of the application so that the user has no or very limited control over the hazardous functionality.

5.9 Execution of external applications

5.9.1 Description

When an application permits a user to have a certain degree of control over external applications executed from the application itself, even when this is not direct, there is a vulnerability of execution of external applications.

There are a number of cases, depending on the function that the application offers:

* *The application enables the user to upload and execute an application in the server. In this case it is possible to enter and run malware directly in the server, thus compromising system security.*

* *The application runs an external application that is selected from an input parameter that can be modified by the user. Depending on the type of control the application carried out, the user has a number of options, from choosing from a list of applications to specifying what the precise application that should be executed is. In the latter case the user can run system orders that can go as far as to compromise server security.*

5.9.2 Example:

```
<input type="text" name="app" value="" />


<%
      Runtime.getRuntime.exec(Request.getParameter(“app”));
%>
```

Example 5.12. Execution of external applications

5.9.3 Consequences

A user that controls external applications executed in the server means a risk to its integrity, confidentiality and availability.

If the user of the system where the applications server is executed has no correct restrictions over the privileges in this environment, the execution of system commands or the inclusion and execution of malware may mean total control over the server.

5.9.4 Risk

High

The most positive scenario consists of access to only a part of the server information. But in many other cases the overall security of the system may be severely compromised.

5.9.5 Solution

The user should never have any kind of control over the applications that can be run in the server.

If the external application to be run is determined by means of a parameter that comes from the user's browser, the web application receiving the parameter should validate it to accept only a small number of valid values.

Besides external applications should be authenticated before being executed to ensure veracity. This can be done for example by checking the MD5 signature.

If the utility for loading and running applications is required, these should be analyzed for virus by specialized applications. However, a user being able to upload executable applications to the server is not recommended.

Finally, it is recommended that the user who executes the applications server has the minimum privileges necessary for the service to run correctly.

5.10 Buffer Overflow

5.10.1 Description

A buffer overflow happens when the data written in a buffer corrupts the information from the memory addresses adjacent to the ones reserved for the buffer. This is usually due to a lack of validation of the input data or poor management of the memory reserves.

In modern programming languages, such as C# or Java with no control over the memory assigned to each variable, these errors are usually caused by the use of libraries (public or private) with which vulnerabilities of this type are associated.

However, these languages also support management of the memory by the programmer. For example, in C#, low level methods of memory management can be used by means of the definition of unsafe contexts.

5.10.2 Example:

```
char buffer[32];
int main(int argc, char **argv)
{
      strcpy(buf, argv[1]);
      printf(buf);
      return 0;
}
```

Example 5.13. Buffer Overflow

5.10.3 Consequences

When modifying the information from the memory addressed adjacent to the buffer, it is theoretically possible to load code into the memory for execution that is different from the application.

The typical consequences of this type of attack are usually execution of system orders, the creation or remote consoles that give access to the user or the possibility of scaling privileges to get more system permits.

5.10.4 Risk

High

If the user can exploit this vulnerability, it can compromise the entire system security, and attack its integrity, confidentiality and availability.

5.10.5 Solution

In high-level languages such as C# or Java, it is preferable to permit the execution environment itself (virtual machine or framework) to be the one to manage application memory rather than permitting the developer to do it directly.

If, for reasons of efficiency or simply out of necessity, direct controls are required over memory management or if languages are used that do not have the capacity for automatic control over them such as C or C++, it is necessary to carry out exhaustive controls over the variables that work directly with the memory (pointers). Some precautions are:

- Checks on the limits of the buffer so that is does not store more data than the memory reserved for this purpose.
- Checks on the numeric variable to control that it does not overflow by exceeding the maximum value of the type of associated data.
- Reject the use of unsafe functions: strcpy, strcat, memcpy, sprintf, scanf, sscanf, etc.
- Use safer functions instead: strncpy, strncat, gets, puts, etc. However it should be remembered that buffer overflows are also possible in most of these functions. an example is to copy a text string that does not end in the character '\0'.
- Do not assume that third party libraries or DLLs are error free.

6 SECURITY DIRECTIVES

This section brings together the knowledge presented up to this point so as to develop a number of security directives to be applied when designing the security mechanisms of an application.

6.1 VALIDATION OF INPUTS AND OUTPUTS

All the inputs of the applications should be validated as restrictively as possible. It should never be assumed that an input is correct and obviate its validation.

Input is understood as meaning any information or data from a reliable (or unreliable) source. These are parameters from web forms, GET request parameters, cookies, HTTP headers, configuration files, data bases, etc.

If the application has a large quantity of inputs from trusted sources, such as a data base, validation of these inputs may be obviated under the responsibility of the entity.

As well as validating the inputs, all the application outputs should be codified to prevent code injection in the interpreters receiving them. The outputs going to HTML interpreters should be codified using HTML codification to prevent Cross Site Scripting, outputs to data bases should be formatted by using collections of parameters to prevent SQL code injections, etc.

6.2 failure prevention

All the security mechanisms should be designed so that when they fail all subsequent requests are rejected.

An example is an authentication system. If the system cannot determine if a user or entity is valid or not, the authentication mechanism should reject the attempt to access it. Another example is a firewall: if it fails, it should reject all traffic received after the breakdown.

6.3 "Keep it Simple"

Designing complex security mechanisms does not guarantee system or environment safety.

In general terms it is preferable to use standards mechanisms that have been exhaustively tested and assessed rather than implementing your own security systems. An example is encryption algorithms, where incorrect implementation may mean the loss of its cryptographic qualities.

6.4 USING/REUSING TRUSTED CONTROLS

Other designers have been confronted before with the problem to be resolved. For this reason it is a good idea to invest time in investigating which is the best solution to avoid a given risk that arises directly from the design of a control mechanism designed by the company itself.

Likewise, once the problem has been resolved it is advisable to document it and announce it to the development team so that if it is necessary to resolve it again, the implemented solution can be reused.

6.5 in-depth defence

Controls cannot be effective 100% of the time.

It is very difficult to predict all the possible situations a security mechanism can be subject to. A good design does not have to take every single situation into consideration, but it is necessary to consider the possibility of error so that if a mechanism is evaded, there is a second one that can prevent the risk.

6.6 AS SAFE AS THE WEAKEST CONTROL

It should never be thought that an application is completely safe because mechanisms that are “impossible” to evade have been implemented in some aspects of the application.

The security of an application is as robust as the weakest control that is implemented, since an attacker firstly analyses the application to detect possible entry points and focuses his efforts on the weakest controls.

6.7 do not base security on concealment

Basing security on concealing information or functions from the user does not usually work.

If the information is only concealed without controlling privileges for their execution, an attempt at direct access by other mechanisms may mean the execution of services or functions by users who do not have access to them.

On the other hand, experience tells us that obfuscation or non-publication of code does not mean that security bugs cannot be discovered by users.

An example is automatic tools that can explore web sites in depth from the web site links themselves or force possible generic routes that are commonly used. The concealment of routes in the web site does not stop these tools from being detected and accessed by users.

6.8 MINIMUM PRIVILEGE

Basing security on concealing information or functions from the user does not usually work.

If the information is only concealed without controlling privileges for their execution, an attempt at direct access by other mechanisms may mean the execution of services or functions by users who do not have access to them.

On the other hand, experience tells us that obfuscation or non-publication of code does not mean that security bugs cannot be discovered by users.

An example is automatic tools that can explore web sites in depth from the web site links themselves or force possible generic routes that are commonly used. The concealment of routes in the web site does not stop these tools from being detected and accessed by users.

6.9 segregation (differentiation by privileges)

To facilitate application administration, organization of users into roles or groups with the same privileges is recommended. This way privileges are granted to roles and not directly to users, which makes it easy to administrate and strictly control the privileges of each user.

7 APPENDIX A: JAVA SUMMARY

The following summary brings together many of the methods and classes that should be reviewed when making a Java application.

7.1 servlets

Servlets are objects that are run in a JEE server or container, and are specially designed to offer dynamic content from a web server. A large number of the methods of these objects are related to the input and output of HTML data, and so should be exhaustively revised.

ServletRequest.getParameterNames
ServletRequest.getParameterValues
ServletRequest.getParameter
ServletRequest.getParameterMap
ServletRequest.getScheme
ServletRequest.getProtocol
ServletRequest.getContentType
ServletRequest.getServerName
ServletRequest.getRemoteAddr
ServletRequest.getRemoteHost
ServletRequest.getRealPath
ServletRequest.getAttribute
ServletRequest.getAttributeNames
ServletRequest.isSecure
ServletRequest.setAttribute
ServletRequest.getInputStream
ServletRequest.getReader
ServletRequest.getServerName
ServletRequest.getLocalName
ServletRequest.getLocalAddr
HttpServletRequest.getAuthType
HttpServletRequest.getRemoteUser
HttpServletRequest.getCookies
HttpServletRequest.getQueryString
HttpServletRequest.getHeaderNames
HttpServletRequest.getHeaders
HttpServletRequest.getUserPrincipal
HttpServletRequest.getMethod
HttpServletRequest.getRequestURI
HttpServletRequest.getRequestURL
HttpServletRequest.getRequestedSessionId
HttpServletResponse.addCookie
HttpServletResponse.addHeader
ServletResponse.getOutputStream
ServletResponse.getWriter
javax.servlet.http.Cookie

Some output methods that should be revised to prevent Cross Site Scripting, HTTP Response Splitting and redirects are:

javax.servlet.ServletOutputStream.print
javax.servlet.jsp.JspWriter.print
java.io.PrintWriter.print
addHeader
setHeader
sendRedirect

HttpServletResponse.sendRedirect	setStatus

7.2 INPUTS AND OUTPUT WITH STREAMS

Some Java functions for working with streams or files are sensitive to attacks from directory traversal or character injection attacks. Some of these functions are:

java.util.zip	ObjectInputStream
java.util.jar	PipedInputStream
FileInputStream	StreamTokenizer
ObjectInputStream	getResourceAsStream
FilterInputStream	java.io.FileReader
PipedInputStream	java.io.FileWriter
SequenceInputStream	java.io.RandomAccessFile
StringBufferInputStream	java.io.File
BufferedReader	java.io.FileOutputStream
ByteArrayInputStream	mkdir
CharArrayReader	renameTo

7.3 SESSION MANAGEMENT

To manage sessions, Java provides some methods that can be used:

getSession	getId
invalídate	

7.4 ACCESS TO DATA BASES

All the accesses to data bases should be revised to find constructions sensitive to code injection attacks. Some examples are:

Jdbc	java.sql.ResultSet.getString

executeQuery	java.sql.ResultSet.getObject
select	java.sql.Statement.executeUpdate
insert	java.sql.Statement.executeQuery
update	java.sql.Statement.execute
delete	java.sql.Statement.addBatch
execute	java.sql.Connection.prepareStatement
executestatement	java.sql.Connection.prepareCall
createStatement	

7.5 Communications encryption (ssl)

Some functions and packets that facilitate the work with SQL connections and that should, therefore, be used in all communications handling sensitive information are:

com.sun.net.ssl	TrustManagerFactory
SSLContext	HttpsURLConnection
SSLSocketFactory	KeyManagerFactory

7.6 event log system

Java provides some libraries to work with event log systems. Some of these libraries are:

java.io.PrintStream.write	qflog
log4j	just4log
jLo	log4Ant
Lumberjack	JDLabAgent
MonoLog	

8 APPENDIX B: EXAMPLES OF REGULAR EXPRESSIONS

This appendix sets out to bring together some of the regular expressions that might be most useful when validating the input data of a standard web form.

* Only alphabetical characters:

  ^[a-zA-Z]+$

  Accepts: “Name”
  Does not accept: “”, “Surname Surname”

  ^[a-zA-Z]*$

  Accepts: “Name”, “”
  Does not accept: “Surname Surname”

  ^[a-zA-Z]+([a-zA-Z]+)?$

  Accepts: “Name”, “Surname of Surname”
  Does not accept: “”, “Surname 1122”

* Only numeric characters (numeric fields):

  ^[0-9]+$

  Accepts: 123456789
  Does not accept: “”, “11111111H”

  ^[0-9]*$

  Accepts: 123456789, “”
  Does not accept: 11111111H

* X code numbers (postal codes, telephone numbers, etc):

  ^[0-9]{5}$

  ^[0-9]{9}$

  …

  Accepts: 12345

  Does not accept: 1234, 123456

* Credit card numbers:

  ^(\d{4}-){3}\d{4}$|^(\d{4}){3}\d{4}$|^\d{16}$

  Accepts: 1111-2323-2312-3434, 1234343425262837, 1111 2323 2312 3434

  Does not accept: 1111 2323 2312-3434, 34323423, 1111-2323-23122-3434

* Dates:

  ^[0-3][0-9]/[0-1][0-9]/[0-9]{4}$

  ^(([0-2][0-9])|(3[0-1]))/((0[0-9])|(1[0-2]))/20[0-9]{2}$

  The second would be more restrictive than the first, but it would still be incomplete, allowing dates such as 29/02/2007, 31/04/2025, etc. For this type of check it is more effective to convert the entered and validated string, for example, with the first expression, in the date format of the programming language.

* Times:

  ^(|(0\d)|(1[0-2])):(([0-5]\d)):(([0-5]\d))\s([AP]M)$

  Accepts: 11:59:59 AM, 00:59:59 PM

  Does not accept: 13:59:59, 11:45:60

* IP addresses:

  ^(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]\d|25[0-5])

  Accepts: 127.0.0.1, 255.255.255.0

  Does not accept: 127.0

* MAC addresses:

  ^([0-9a-fA-F]{2}:){5}[0-9a-fA-F]{2}$

  Accepts: 00:11:22:33:44:55:ab

  Does not accept: 00:11

* Electronic mail addresses:

  ^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$

  Accepts: name.surname@domain.com, name-surname@127.0.0.1

  Does not accept: this_is_nogood@domain

* URL addresses:

  ^(http|ftp|https):\/\/[\w\-_]+(\.[\w\-_]+)+([\w\-\.,@?^=%&:/~\+#]*[\w\-\@?^=%&/~\+#])?

  Accepts: http://regxlib.com/Default.aspx, https://electronics.cnet.com/electronics/0-6342366-8-8994967-1.html

  Does not accept: www.yahoo.com

9 REFERENCES

* *"Code Review Guide, V1.1"*. OWASP, 2008
* M. Curphey, D. Endler, W. Hau, S. Taylor, A. Russel, G. McKenna, R. Parke, K. McLaughlin. *"A Guide to Building Secure Web Applications"*. OWASP, 2002
* B. Chess, G. McGraw. *"Static Análisis for Security"*. IEEE Security & Privacy, 2004
* http://msdn.microsoft.com
* http://www.owasp.org
* http://www.regxlib.com
* https://www.securecoding.cert.org
* http://cwe.mitre.org/