

Case study:

Secure development of web applications

Table of Contents

Introduction	3
The importance of web application security	4
Security Architecture Overview	5
Secure Database Design Considerations.....	6
Auto Generation of Application Layers.....	7
Auto Generated Data Access Layer.....	7
Auto Generated Presentation Layer	8
Auto generation of Validation Layers	8
Database Input Validation Layer.....	8
Client Side Input Validation	10
Request Inspection	11
General Protection.....	11
File-upload Handling	12
Denial of Service Protection.....	12
Cross Site Scripting Protection.....	12
SQL Injection Protection	13
Incident Response.....	13
Member Authentication	14
Password Policy.....	14
Password Cracking Sample	14
Multifactor Authentication	15
Conclusion.....	16

Introduction

We all know Google, Facebook, Twitter, Amazon and all the other Internet giants, and common for them all is that they base their business around large web applications. To earn profit and keep their business running, they need users to visit their web sites, share information, buy products and use the web site as a communication platform. Facebook made a record setting \$1,458 Billion in Q1 2013¹, with \$1.25 billion coming from advertising alone, which makes them a good example on how user data can be collected and sold as a service to advertising customers. But Facebook have security vulnerabilities, and with more than 60 new vulnerabilities found on each year². What makes us think that all the other site on the Internet are different? The truth is that most websites on the Internet are vulnerable to at least one security flaw, which goes to show that we must be doing something wrong when it comes to web development.

In this case study, I will build a web application with focus on security and privacy. It should be resistant to most known attack techniques, while still providing flexibility for the users, in the form of convenience and functionality. And what is better than building this web application for a hacker group? I'll take a look at some of the common mistakes that is made when developing web applications using the traditional methods, as well as show another approach to web development through auto-generation of layers using secure templates, as well as a runtime extensible security framework to protect against - and report - common attacks.

¹ <http://investor.fb.com/releasedetail.cfm?ReleaseID=761090>

² <https://www.facebook.com/whitehat/thanks/>

The importance of web application security

Web applications are on the rise, and business are moving their infrastructure to cloud and mobile platforms to reach their customers, sell products, provide new services and share information. More importantly, hospitals, social services and other government run institutions, are also moving their systems to a more high tech cloud based infrastructure. Denmark has recently invested 40 billion³ kr. to shut down the old hospitals, and creating new so called super hospitals, where IT is in focus. Denmark is moving to a more digital platform, take for example Borger.dk, where citizens of Denmark can log in and view their hospital and doctor visits, information about their job and insurance. Other examples are Skat.dk, which holds financial records and pensionsinfo.dk, which have pension data for all citizens. It is no secret that most of the information about Danish citizens are now stored online, in large databases behind a web application. But just how secure are those web applications?

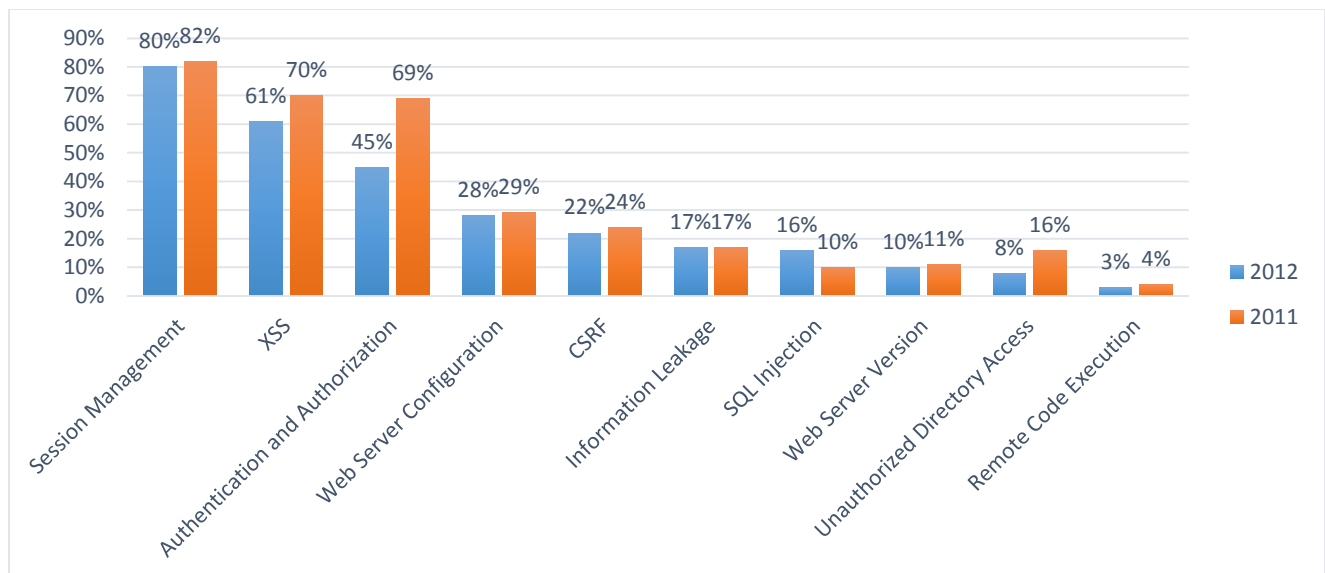


Figure 1: Security vulnerability trends 2012 according to Cenzic

According to one vulnerability report⁴, 99% of all tested web applications had security vulnerabilities, with Cross Site Scripting (XSS) appearing in 61% of all tested web application in 2012. Another report⁵ found that 86% of all web applications had at least one vulnerability, and 43% of the tested web applications had XSS. If any of the sites mentioned above hold such a vulnerability, it could have catastrophic consequences as the attackers would have full access to the personal data of millions, which in turn could be used to steal the identity of any citizen. A study⁶ from 2012 found that each of the 56 participating companies share an expense related to cybercrime of \$8.9 million per year. So not only do most companies have a vulnerability in their web application, some also have large expenses related to incident management.

³ [http://www.investindk.com/News-and-events/News/2013/Denmark-Invests-\\$7-billion-in-Super-Hospitals-for-the-Future](http://www.investindk.com/News-and-events/News/2013/Denmark-Invests-$7-billion-in-Super-Hospitals-for-the-Future)

⁴ <https://info.cenzic.com/2013-Application-Security-Trends-Report.html>

⁵ <https://www.whitehatsec.com/resource/stats.html>

⁶ http://www.ponemon.org/local/upload/file/2012_US_Cost_of_Cyber_Crime_Study_FINAL6%20.pdf

Security Architecture Overview

Most web applications build on defensive programming⁷, and rely on input filtering and encoding to protect against security vulnerabilities in their underlying architecture and business logic. They often weave in the validation code with the business logic; use 3 tier models or MVC that contain validation code in each of the layers, in order to protect against malicious users.

I opted for a solution with a modularized extensible security framework for small to medium web applications, which is based on a three part validation layer. It extends from the client, to the business logic and down to the database. Compared to ordinary models, this one is modularized and can be extended at runtime. The way it shims in between the client and application, and in between the data layer and database, means it is compatible with current Separation of Concerns (SoC) patterns like Model View Controller (MVC) and multitier data models.

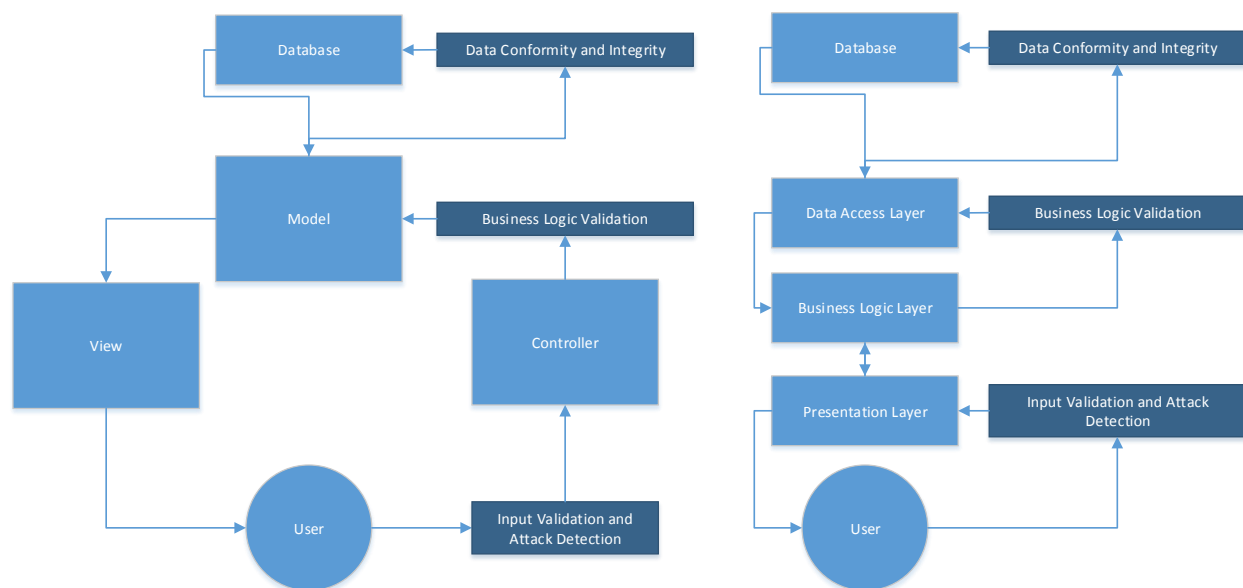


Figure 2: Placement of validation and detection layers in a MVC and 3 tier model

I choose this method as it would separate the validation logic from the application layers. More importantly, it would enable me to auto generate much of the validation layers from a metadata model and database constraints.

⁷ http://en.wikipedia.org/wiki/Defensive_programming

Secure Database Design Considerations

Many vulnerabilities are based around poor database design choices. One example is Facebook that had a privacy incident back in 2010, where 171 million⁸ Facebook profiles were downloaded and distributed. This was possible because the identification numbers used in the database design were incremental, and therefore possible to predict.

To protect against common security and privacy related issues, I created a list of security aware design guidelines for the database:

Use unique random identifiers as the primary key for data that should not be public and predicable.

This design guideline contradicts common best practice database design guidelines, where you are advised to use natural keys⁹ formed from an already existing attribute on the table. However, in a secure database, it would be a security risk to use predictable or publicly available attributes as the primary key. Data tables like user profiles should never be predicable or publicly available as a list, so to protect the identity and privacy of the users, they should have a unique pseudo random identification number as the primary key. This protects against mass download of profiles, and hackers that try to guess the id of privileged users.

Avoid Unicode support for data that does not use Unicode.

Many social engineering attacks rely on the fact that users can create usernames that resembles legitimate user names or emails using Unicode. Urls are also a concern, as they can trick users into clicking a link that closely resembles a legitimate link.

Example: The character / and / closely resembles each other, but the first one is Unicode character U+2044, also called a fraction slash. Due to support for Internationalized Domain Names (IDN) in browsers, malicious users can trick users into visiting fake websites that contain a fraction slash instead of a real slash.

Use small data type sizes

Many database designers choose a large size for data types by default, instead of considering the actual data the field will contain. There is no need for a varchar(MAX) size for usernames in the first place, so usernames should be confined to a reasonable length. It also gives a variety of security problems and can be misused in social engineering attacks.

Example: A user can create a username like this “MyUser
MyUser2”

It might not be immediately obvious, but the user inserted a lot of spaces, and now appear as having two usernames, due to the website wrapping the blank spaces between in the username. XSS and other kinds of injection attacks are also more likely to be successful if data fields are not restricted in length.

⁸ <http://www.skullsecurity.org/blog/2010/return-of-the-facebook-snatchers>

⁹ http://en.wikipedia.org/wiki/Natural_key

Use fixed size fields and support auto-generated data

Most database designers prefer to save the original user data submitted to a website. However, whenever possible, one should consider using auto-generated names instead. Files supplied by the users should be renamed to a random fixed-length name and then saved on the server. If needed, the original filename can be normalized using a whitelist, and then saved to the database.

Example: A user uploads a file named "myLink" to a website. Note that all the characters in that filename is valid on UNIX based systems. Whenever someone is viewing the filename on the website, they immediately get redirected to www.evilsite.com

The fixed length should be 10 characters or more to prevent brute force attacks and mass download of files from the site.

Always use data types that fits the data best

Zip codes, phone numbers and social security numbers are often saved as strings with an arbitrary field length. Web applications should constrain the input in such a way that the data is stored in a uniform format in the database, with a data type that fits the type of data. Instead of just putting "+45 12-34-56-78" in the database as a phone number, one should constrain it to 2 digit area code and 8 digit phone number.

Auto Generation of Application Layers

Instead of designing the application with defensive coding techniques and weaving contracts into the code, I opted for a solution that auto generates the application based on a metadata model. Using this technique, I would have to maintain a small file which defines the input/output capabilities of the application. If the metadata is correct and the application acts accordingly, it should make the application more secure compared other solutions. Many web applications have an issue with input validation, where developers forget a field or a whole page, because validation code has to be written manually. In this case, the input validation can be generated automatically, and it will be applied to all fields in the application.

Auto Generated Data Access Layer

The data access layer is auto generated using a well-known object relational mapping tool called Entity Framework (EF). It is maintained by Microsoft and released as an open source project¹⁰ on Codeplex. It has the ability to create Plain Old CLR Object (POCO) classes which then can be used in a repository design pattern. The classes are small and easily extended with custom code as they are marked as partial¹¹ by default. It also enables writing custom validation code for an entity, which is persisted even when the class is auto-generated again.

The classes are generated from a template that have properties with a data type that is compatible with the data type from the database. This means a smalldatetime from the database will result in a DateTime in .NET. However, because smalldatetime only can represent dates from 1900-01-01 through 2079-06-06, and .NET DateTime can represent 0000-01-01 through 9999-12-31, it is possible to put an

¹⁰ <http://entityframework.codeplex.com/>

¹¹ <http://msdn.microsoft.com/en-us/library/wa80x488%28v=vs.80%29.aspx>

invalid DateTime into a smalldatetime. To protect against this, the validation layer makes sure that all dates that are sent to the database, are in the accepted range, which at the same time should be customized according to the business logic of the application.

Auto Generated Presentation Layer

To harness the full power of an auto generated data access layer, I opted for a research project called ASP.net Dynamic Data, which is now released as a part of the .NET framework. The whole idea behind Dynamic Data is to read a data model like Entity Framework, and then auto generate scaffolds that give the ability to perform Create, Retrieve, Update, Delete, List (CRUDL) operations on the entity. However, the scaffolds are not really useable with their standard templates, so I created my own which are more flexible and secure.

To reduce the attack surface, the first change was to combine the insert, update and details templates into a single template. As the template is going to be used to insert data into the database, I made sure that all components used in the template were securely encoded and never sent data to the database without going to the validation layer first. Next I locked down the templates in such a way that entities can't be deleted, inserted or edited by default, and instead would have to be marked as such in the metadata model. This way I don't grant the ability to delete users, logs or other entities without explicitly marking it as capable of being deleted.

Auto generation of Validation Layers

Creating application input validation is often tedious, and especially if the validation logic needs to span multiple layers, such as going from the business layer, and up to the client. Take for example an event system with the ability to schedule party events. The date of the event needs to be validated, and the business logic dictates that the date must not be before the current date, and also not after 1 month from the current date (into the future). This logic needs to be implemented into the business layer, but also into the presentation layer for convenience, and also so that the client does not submit the event without a valid date.

In an ordinary system, you would have to write the backend code to validate the date, and then write the JavaScript code to validate the date in the frontend. If a change happens to the backend, you would also need to make the same change to the frontend. Writing and maintaining this kind of code is hard work and often gets overlooked or completely skipped.

However, most of this kind of validation can be defined once in a metadata file, and then auto generated in both the data access, business and presentation layers. To achieve this, I opted to build the web application with runtime auto generated templates, which have compile time validation based on data types (int, string, date), data kinds (html, url, SSN) from the database and a separate metadata file. At the same time, it is easily extended with custom validation code, such as regular expressions, that then gets propagated to the application layers.

Database Input Validation Layer

The data access layer itself was auto generated by the Entity Framework based on a data model of the database. However, this layer contains no validation of the entities, so I had to come up with a solution that could read the metadata from the database, and create code that would validate entities against it.

Entity Framework comes with the ability¹² of reading a metadata class with validation code, but it have no way of auto generating the validation code based on constrains from the database, so I developed¹³ a T4 code generation template using data annotations¹⁴ that reads the metadata of the database and then tries to create the correct validation code, based on data types and the name of the fields. It is easily extended with custom validation code, and when a new table is added to the database, it gets validation applied after running the template again.

Take for example the following structure called Member in the database:

Column Name	Data Type	Allow Nulls
UserId	Uniqueidentifier	False
Name	Varchar(50)	False
Email	Varchar(100)	False

Based on the data types, and the fact that all the columns require data to be present, we can make some assumptions in the metadata model. Entity Framework does a good job of mapping database data types to CLR data types, so in the metadata model auto-generation template, we have to read out the CLR type on from the entity property in EF, and write it into the metadata model. Next we read out of if the property allow nulls or not. If it does not allow nulls, like in the example above, we mark the metadata model with "[Required]" to indicate the field is required.

By guessing the kind of data that is contained in a property, we can put additional constrains on the data such as a specific data format. The auto-generation template tries to guess the data by looking at the name of the property, and if it matches any known name, such as "email", it will add an "[DataType(DataType.Email)]" attribute to the property. This way data entered in the field have to comply with RFC2832¹⁵ if there is no other regular expression specified.

```
[MetadataType(typeof(MemberMetadata))]  
[ReadOnly(true)]  
public partial class Member  
{  
    [DisplayColumn("Name")]  
    internal sealed class MemberMetadata  
    {  
        [Required(ErrorMessage = "User Id is required")]  
        [DisplayName("User Id")]  
        public GUID UserId { get; set; }  
  
        [Required(ErrorMessage = "Name is required")]  
        [StringLength(50, MinimumLength = 1)]  
        public String Name { get; set; }  
  
        [RegularExpression(@"\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b")]  
        [StringLength(100, MinimumLength = 1)]  
        [DataType(DataType.Email)]  
        public String Email { get; set; }  
    }  
}
```

¹² <http://msdn.microsoft.com/en-us/data/gg193959.aspx>

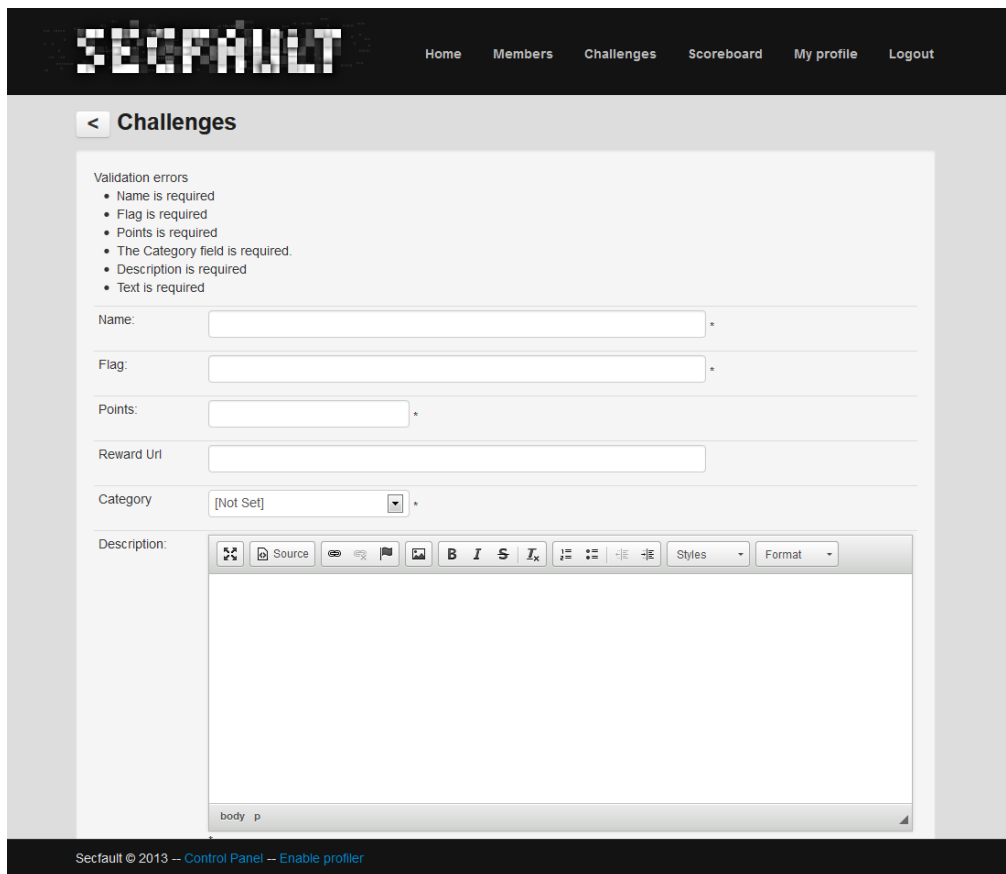
¹³ <https://github.com/Genbox/EFValidationTemplate>

¹⁴ <http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx>

¹⁵ <http://tools.ietf.org/html/rfc2822#section-3.4.1>

Client Side Input Validation

Client side validation is done using JavaScript, and since we already have defined a validation layer in the DAL, we can replicate the validation layer in the presentation layer. Luckily, ASP.net 4.0 Dynamic Data supports reading Entity Framework data model validation code, and convert it to the appropriate JavaScript using ASP.net validation controls. They also have the ability to transfer error messages from the validation layer to the presentation layer, which makes it easy to maintain several language resource files, and keeps all the error messages inside the metadata model.



The screenshot shows the 'Challenges' page of the Secfault application. At the top, there is a navigation bar with links: Home, Members, Challenges, Scoreboard, My profile, and Logout. Below the navigation bar, the page title is '< Challenges'. The main content area displays a form for creating a challenge. Above the form, there is a section titled 'Validation errors' with a bulleted list of errors: 'Name is required', 'Flag is required', 'Points is required', 'The Category field is required.', 'Description is required', and 'Text is required'. The form fields are: 'Name:' (text input), 'Flag:' (text input), 'Points:' (text input), 'Reward Url' (text input), 'Category' (dropdown menu showing '[Not Set]'), and 'Description:' (rich text editor). The rich text editor has a toolbar with icons for source, link, unlink, image, bold, italic, strikethrough, bulleted list, numbered list, indent, and styles, along with a 'Format' dropdown. The footer of the page contains the text 'Secfault © 2013 -- Control Panel -- Enable profiler'.

Figure 3: Client side validation with error messages from the metadata model.

Request Inspection

One of the most important security concepts, that often gets overlooked, is the logging and reporting of malicious activity. One of the reasons is that there are few easy to use pluggable security frameworks that integrate with applications. Those whom choose to use such a framework, usually install a WAF¹⁶, which integrate on the service level in the OS, and are therefore not usable on shared hosting systems.

Since the web application in this case is going to be run in a shared hosting environment, and we need extra strong security without any false positives, we decided to implement Security Runtime Engine from Microsoft Web Protection Library¹⁷. Please note that the request validation in this case is only used for added security and reporting, as WAF and other security frameworks is not enough¹⁸ and more aggressive fundamental security features have been implemented.

General Protection

Some browsers support HTTP headers sent from the web application to increase security. The headers are not well known, and thus is not used on that many sites. I have listed those that are implemented in this web application:

X-FRAME-OPTIONS: DENY

To protect against click-jacking, Firefox and IE support a header called X-FRAME-OPTIONS¹⁹, and it set to DENY. This means that whenever an external site tries to embed the page in a frame or iframe, it gets denied by the browser. It can be set to SAMEORIGIN to allow the site to use parts of itself in a frame, but that is not the case here. Usually this header is set in the web server configuration, but as this is a shared hosting with no access to the configuration, I opted to enforce this in the web application.

X-DOWNLOAD-OPTIONS: noopen

Some vulnerabilities rely on the fact that when a file is downloaded from the server, and the user clicks open, the file is executed in the security context of the browser. To prevent this kind of attack Internet Explorer reads a flag called X-DOWNLOAD-OPTIONS²⁰, and when it is set to noopen, all download dialogs have their 'open' button removed.

Set-Cookie: HttpOnly

This HTTP header makes sure that cookies are can't be read from JavaScript. It is a very efficient method of protecting against cookie stealing from an XSS attack. The web application enforce the HttpOnly flag on all cookies sent from the server to the client.

Fake Viewstate

While this is not a HTTP header, it is an integral part of ASP.net. The viewstate is an XML document that is sent from the server to the client on the first visit, and in all subsequent clicks, the viewstate is passed from the client to the server.

¹⁶ http://en.wikipedia.org/wiki/Application_firewall#Specialized_application_firewalls

¹⁷ <http://wpl.codeplex.com/>

¹⁸ <http://www.acunetix.com/blog/news/implementing-a-web-application-firewall-only-is-not-enough-to-secure-web-applications/>

¹⁹ <https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options>

²⁰ <http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx>

File-upload Handling

Many sites are compromised from files uploaded to the site. It might be a profile image chosen by the users, or a file sharing service. They are compromised by using so called “shells” which are small applications/scripts that can be executed on the webserver. They used to be prevalent and one of the primary attack methods for taking over servers running PHP using Remote/Local File Inclusion (R/LFI) vulnerabilities. To protect against those kinds of attacks, I implemented multiple deep inspection checks to make sure that profile images uploaded to the server is indeed images, and not shells. Another protection mechanism is that images are 10 character long random filenames, and can only have the .png, .jpg or .jpeg extensions on the server. Even if an attacker managed to upload a shell to the server, it would not get executed due to the forced extension, and there is no file inclusion available on the server as that is managed automatically by ASP.net Master Pages.

Denial of Service Protection

Usually there is no way for a web application to protect itself from Denial of Service (DoS) attacks. DoS attacks work by making a lot of requests to a page on the web server, which in turn exhausts the resources of the server. Since most web sites, as well as this one, have external data sources, they get DoS attacked too. In order to protect against DoS attacks, I made sure that once an IP had made more than 100 requests within a short amount of time, it will get rate limited and presented with a very lightweight page. Not only does this protect our external data sources, but it also helps against brute force attacks against something like the login page.

To truly protect against DoS attacks, we need to block the incoming requests before they ever reach the server. To do this, we opted to use the CloudFlare service²¹ which include IP rate limiting and DoS protection. With several layers of protection, we are at least minimizing the impact of a DoS attack.

Cross Site Scripting Protection

The site have implemented aggressive XSS protection that works in a two way fashion: Input & output. For input, we detect special characters used for XSS such as < ' / and other and block them before they ever reach the web application code. Places where such characters should be allowed, such as password fields, the detection is disabled, and the input is encoded instead.

The XSS output protection is a failsafe that makes sure that every ASP.net control on the site encodes its output. This means that in the worst case scenario, where an attacker manages to insert a XSS into the application, it will not be executed by the other users.

In the few cases where we would like HTML to be executed, like the news on the front page, a small number of HTML tags are allowed, and only by the Admin role on the server.

²¹ <http://www.cloudflare.com/features-security>

SQL Injection Protection

One of the most dangerous attacks is SQL injection²². One of the reasons is that it can be used to extract data directly from the database, by using the web application itself. To protect against it, one should use SQL parameters for all SQL queries, which is also what we do. Actually, this is done automatically by the Entity Framework, so we don't have to worry about SQL injections. However, we would still like to report any attempts to execute SQL on the webpage, so as part of the request inspection, a T-SQL parser tries to execute the input as SQL, and if it succeeds, it gets reported as a severe incident, and the request gets blocked. Using a real SQL parser instead of looking for special symbols lowers the amount of false positives that gets reported, and we can block the request more reliably.

Incident Response

If an attacker inputs special characters into a field that should not contain such characters, it gets reported as an incident. The same thing happens to any field that contains a full or partial XSS or SQL injection, or tries to tamper with the view state. The incidents are then collected into an incident manager that decides when it is time to rate limit, ban or accept an incoming connection.

With 5 incidents, the IP gets banned and the web application admin gets an email. This means that any automated web vulnerability tools will quickly get banned and render the tool useless. Take for example the case where an attacker tries to brute force the login form in the web application, or perform a XSS attack, it gets reported as an incident, and the user is banned. Most attacks are usually carried out using a proxy network, and since our members are geographically centered in and around Denmark, we implemented a large list of open proxies, as well as Tor exit nodes²³ to block any attackers that tunnel their attack through a proxy network. If the website detects the use of a proxy, it will report it as an incident and block the connection.

Since the security framework is runtime extensible, it is possible to create a module that reports the incident by SMS or to a third party database, without ever having to shut down the application. Further request inspections and protection against attack vectors such as new kind of attacks can also be plugged in at runtime, which makes it easy to work with and more flexible than other solutions that require downtime for maintenance.

²² https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#OWASP_Top_10_for_2013

²³ <http://proxy.org/tor.shtml>

Member Authentication

To protect the member's passwords against wordlist and brute force cracking, I implemented secure password hashing²⁴ using PBKDF-2²⁵ with 4096 HMAC-SHA1²⁶ iterations, 512 bit key and 1024 bit salt.

Password Policy

To make sure that passwords can't be guessed and cracked in a short amount of time, we enforce a strong password policy with no maximum size or symbol limit. The requirements are the following:

- At least 10 characters
- At least one symbol
- At least one digit
- At least one upper case
- At least one lower case

To facilitate strong passwords, users get a visual feedback of their password strength. However, it is based on a preferred password length of 20 to encourage users to use a stronger password than the password policy.

Password Cracking Sample

With this password policy, and the hashing scheme with PBKDF-2, we can calculate how long it would take to crack a single password:

We have 10 characters of numbers, upper, lower and symbols:

0-9: 10 combinations

a-å: 29 combinations

A-Å: 29 combinations

Symbols: ~11 combinations

Total: ~79 combinations

$$79^1 + 79^2 + 79^3 + 79^4 + 79^5 + 79^6 + 79^7 + 79^8 + 79^9 + 79^{10} = 9589664237532319600$$

With a high-end GPU (in 2013), one can achieve speeds of around ~30.000 hashes each second.

$$\frac{9589664237532319600}{30000} = 319655474584410$$
$$\frac{319655474584410}{31536000} = 10.13 * 10^6$$

That means it would take approximately 10 million years to brute force a hash with this hashing scheme and password policy.

²⁴ <http://crackstation.net/hashing-security.htm>

²⁵ <http://en.wikipedia.org/wiki/PBKDF2>

²⁶ http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

Multifactor Authentication

To improve security for members that wish to do so, we have implemented an optional multifactor authentication as described in RFC 6238²⁷. The code²⁸ is open source and have been checked against Google Authenticator²⁹ as the reference. The timeframe is 30 seconds until the password changes. For added security, the 2 factor secret is stored in the database as an encrypted blob. It is encrypted with a 256 bit PBKDF-2 key with 1024 iterations of HMAC-SHA1; the key is then used as an encryption key for AES-256. When users activate the 2 factor authentication, they are given the 128 byte secret key in the form of a QR code, as it would be impractical to enter 128 characters into a mobile phone. The secret is sent to Google Chart API using HTTPS to prevent Man in the Middle (MitM) attacks.



Each time the member deactivate and reactivate the 2 factor authentication, a new secret is generated and the mobile device of the member needs to be updated. Randomizing the secret each time tighten security a little bit compared to reusing the same secret, as the member now has the ability to change secret, if his mobile device have been compromised.

²⁷ <http://tools.ietf.org/html/rfc6238>

²⁸ <https://github.com/rickbassham/two-factor>

²⁹ <http://code.google.com/p/google-authenticator/>

Conclusion

High security template based web development is a secure and flexible web development method that reduce the time used on some of the more tedious areas of a web application. Client side validation, data presentation and streamlined functionality across pages are done automatically by the templates, and provided that the metadata model has been defined correctly, and have the correct validation logic, all the web pages in the website till be covered. Small to medium websites can employ this kind of technology, at least to parts of the website, to further reduce the attack surface and protect their application.

It is important to implement protection against common attack vectors and reporting them such that the administrator of the site knows about attacks against the web application, and to facilitate some kind of incident response.