ANDERSON SUSSUMU SONEHARA

VULNERABILIDADES EM APLICAÇÕES ASP.NET

Trabalho de conclusão de curso apresentado no Curso de Especialização em Desenvolvimento de Softwares em Mercados Internacionais, na Universidade Federal do Paraná - UFPR.

Orientador: Prof. Dr. Celso Yoshikazu Ishida

CURITIBA
2013

UNIVERSIDADE FEDERAL DO PARANÁ
DEPARTAMENTO DE CIÊNCIA E GESTÃO DA INFORMAÇÃO
Curso de Especialização em
Desenvolvimento de Software em Mercados Internacionais

TERMO DE APROVAÇÃO

VULNERABILIDADES EM APLICAÇÕES ASP.NET

por

ANDERSON SUSSUMU SONEHARA

Este Trabalho de Conclusão de Curso (TCC) foi apresentado(a) em 14 de dezembro de 2012 como requisito parcial para a obtenção do título de especialista em Desenvolvimento de Software em Mercados Internacionais. O(a) candidato(a) foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Dr. Celso Yoshikazu Ishida
Prof.(a) Orientador(a)

Prof. Celso Yoshikazu Ishida
Coord. CEDSMI
Mat 201798-3314966

Dr. Ricardo Mendes Jr
Membro titular

Ricardo Mendes Jr.
Mat. 8236
E-mail: mendesjr@ufpr.br

Membro titular

Dr. Celso Yoshikazu Ishida
Coordenador do CEDSMI - UFPR

Prof. Celso Yoshikazu Ishida
Coord. CEDSMI
Mat 201798-3314966

Vulnerabilidades em aplicações ASP.NET

Anderson S. Sonehara. CEDSMI[1] - UFPR

Abstract

This article aims to present the main types of vulnerabilities in web applications development using .NET. There are various types of vulnerabilities which still exploited frequently. Some vulnerabilities can be easily corrected, others can be avoided by the developer during coding. It is described the *SQL Injection*, the *Cross Site Scripting (XSS*) and the *Cross Site Request Forgery* (CSRF).

Keywords: vulnerability, web, sql injection, cross-site scripting (XSS), cross-site request forgery (CSRF), hash, salt, cryptography.

Introdução

Segurança tem sido um tema muito abordado em palestras, artigos e fóruns. Contudo, na prática, às vezes pelo cronograma apertado e baixo orçamento, as empresas acabam deixando de lado a segurança. Uma pergunta que surge: O que sua empresa tem feito para melhorar a segurança do seu sistema?

As boas práticas de desenvolvimento de software minimizam o risco de algum tipo de ataque, mas algumas vulnerabilidades ainda podem ser exploradas, tais como: o *SQL Injection*, *Cross Site Scripting (XSS), Cross Site Request Forgery* (CSRF) e adulteração de dados.

As vulnerabilidades que serão apresentadas podem acontecer em qualquer ambiente de desenvolvimento – ASP.NET, PHP ou Java - porém cada tecnologia apresentará uma solução diferente. Este artigo estará apresentando somente as soluções para o ambiente.NET, e os exemplos mostram códigos em C# ou ASP.NET.

O sítio "The Open Web Application Security Project" (https://www.owasp.org) publicou os dez principais riscos de segurança em aplicações[2], dentre elas estão o *SQL Injection, Cross-Site Scripting*(XSS) *e Cross-Site Request Forgery* que serão comentados neste artigo. Em 2010, o XSS ficou entre as vulnerabilidades mais perigosas segundo o sítio http://cwe.mitre.org.

---

[1] CEDSMI – Curso de Especialização em Desenvolvimento de Software em Mercados Internacionais
[2] https://www.owasp.org/index.php/OWASP_Top_Ten

No período entre janeiro e março de 2013 houve um aumento de 132% em ataques CSRF, segundo o sítio www.net-security.org. Ataques *Cross-Site Scripting* (XSS) tiveram um aumento de 87% em relação ao primeiro trimestre de 2012.

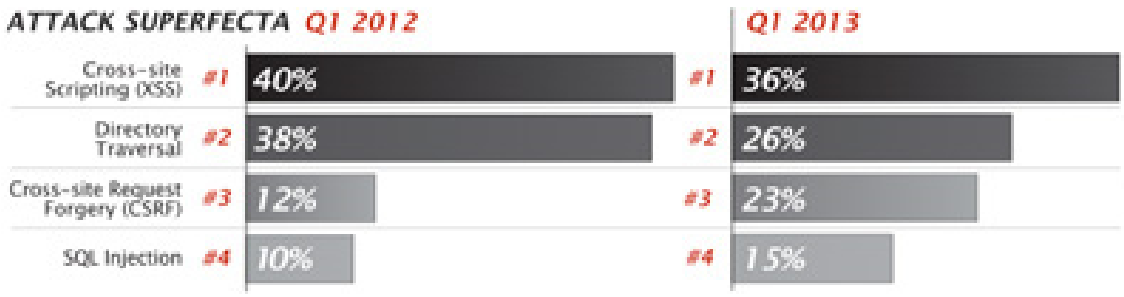

Gráfico 1 – Crescimento de ataques CSRF em relação ao primeiro quadrimestre de 2012 (Fonte: http://www.net-security.org/secworld.php?id=14794)

No gráfico abaixo, extraído do sítio *WhiteHat Security*[3], é apresentado as principais classes de vulnerabilidades de 2012. Observa-se que o CSRF aparece em 26% das aplicações web, o que enfatiza a necessidade das empresas se prevenirem contra esse tipo de ataque.

Esta pesquisa foi realizada em mais de 7 mil sites, entre mais de 500 organizações de diversas marcas e áreas do mundo.

[3] Empresa americana que oferece soluções de segurança em sites.

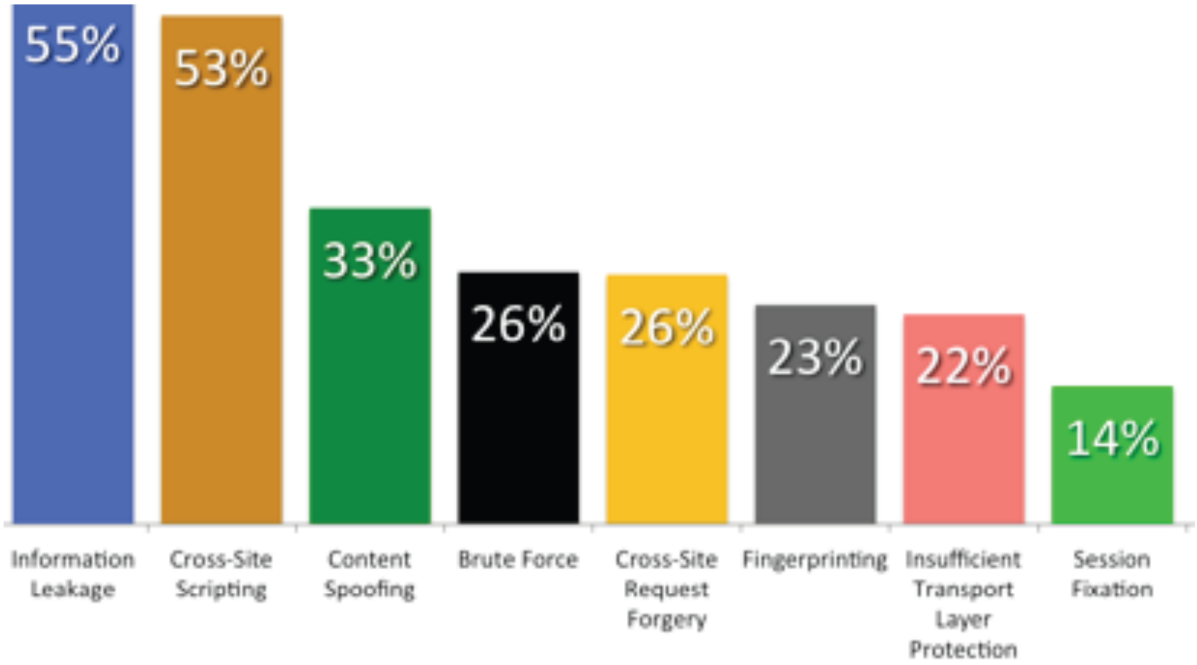

Gráfico 2 – Principais vulnerabilidades em 2012 (Fonte: www.whitehatsec.com)

A seguir, será detalhado o SQL Injection, seu conceito, exemplos e prevenção.

1 SQL Injection

SQL Injection é um tipo de falha de segurança em que o invasor adiciona códigos SQL num campo de formulário web para poder ter acesso a dados ou realizar alterações de informações no banco de dados. Neste tipo de falha o comando SQL é executado para que se faça alguma ação no banco de dados. O SQL Injection normalmente acontece nos formulários de autenticação, quando o usuário digita nome e senha. É nesse momento que se pode inserir comandos SQL na query original. Campo de pesquisa é outra situação onde também é possível explorar esta vulnerabilidade.

O código em C# a seguir exemplifica como o SQL Injection ocorre:

```
query = "SELECT * FROM usuarios WHERE nome = '" + nomeUsuario + "';"
```

Onde, ‘nomeUsuario’ é uma variável que pode ser obtida, por exemplo, diretamente de um campo TextBox digitado pelo usuário.

O código SQL acima foi desenvolvido para retornar um usuário específico através do seu nome, acessando a tabela usuários. Entretanto, se a variável “nomeUsuario” for alterada por um usuário malicioso, a query original pode fazer mais do que realmente deveria.

Por exemplo, vamos alterar a variável “nomeUsuario” para:

' or '1'='1’

O código do SQL final poderá ficar assim:

```
SELECT * FROM usuarios WHERE nomeUsuario = '' OR '1'='1';
```

Se o código fosse realmente executado, este exemplo poderia forçar a seleção de usuário válido mesmo não sabendo nome do usuário correto, pois a premissa OR '1'='1' sempre será verdadeira.

Também é possível comentar o restante do código original da query:

' or '1'='1’ --

Caso o usuário digitar “a'; DROP TABLE usuarios;” (valor que irá para a variável “nomeUsuario”) o código do SQL final poderá ficar assim:

```
SELECT * FROM usuarios WHERE nomeUsuario = 'a'; DROP TABLE usuarios;
```

E, se o código for executado, irá apagar a tabela “usuários”.

Assim com foi inserido o código ‘drop’, poderia ser inserido outros como ‘update’, ‘insert’, ‘delete’ ou, até mesmo, queries mais complexas.

Existem diversas maneiras de prevenir o SQL Injection, entre elas podemos destacar:

- Uso de parâmetros
- Uso de Stored Procedure
- Evitar uso do SQL InLine
- Não concatenar Strings com comandos SQL
- Use Regex
- Prefira o uso de **ORM´s (Entity Framework, nHibernate)**

Parâmetros: Garanta que todos os dados passados para sua query principal seja realizada através de parâmetros.

Stored Procedures: Sempre que possível utilize Stored Procedures, por diversos motivos: desempenho, manutenibilidade, segurança, etc.

Evite SQL Inline: Evite código SQL misturado com o código fonte.

Não concatene strings: Por ex: var sql = String.Format(“select * from tabela where {0}”, condição)

Regex: Sempre que possível utilize expressões regulares para filtrar os parâmetros de entrada. Por exemplo: strPesquisa = Regex.Replace(strPesquisa, "[^a-zA-Z0-9]", "");

ORM´s (Entity Framework, nHibernate): Utilize ORM´s sempre que possível, pois estes utilizam passagem por parâmetros e evitam o SQL Injection.

Exemplo, mais seguro, utilizando Stored Procedure com passagem de parâmetros:

```
SqlCommand cmd = new SqlCommand();
string query = @"Proc_PesquisaProduto";
cmd.Parameters.Add(new SqlParameter("@NomeProduto", strPesquisa));
cmd.CommandType = CommandType.StoredProcedure;
cmd.CommandText = query;
```

2 Cross-Site Scripting (XSS)

Cross-site Scripting(XSS) é um tipo de vulnerabilidade normalmente encontrado em aplicações web, onde o atacante pode injetar um script nas páginas acessadas por outros usuários. Existem dois tipos de XSS:

Não persistente: XSS não persistente acontece quando o usuário acessa diretamente uma página que contém um código ou script nocivo.

Persistente: XSS persistente acontece quando a página permite que usuários postem ou submetam informações para serem armazenadas no banco de dados e posteriormente exibidas para os próximos visitantes do site. Por exemplo: fóruns e blogs.

Quando o site está exposto a essa vulnerabilidade vários códigos nocivos podem ser executados, tais como: executar blocos de códigos Javascript ou jQuery e consequentemente criar ou acessar qualquer elemento da página, capturar eventos de Javascript (onMouseOver, onClick, etc), alterar cookies e muito mais.

Um teste bem básico para um blog: verificar se é possível postar o seguinte código:

```
<script> alert('teste')</script>
```

Se surgir uma tela de alerta, este é um dos indícios de que este site é vulnerável ao XSS.

A prevenção do XSS no .NET é fácil. Utilize expressões regulares para evitar que tags HTML ou blocos de código Javascript sejam submetidos. Caso necessite submeter caracteres que possam ser interpretados pelo navegados como tags, utilize o comando Server.HtmlEncode do C#. Por último, nas páginas .aspx, em seu cabeçalho deixe sempre a propriedade ValidadeRequest = True, assim o próprio framework bloqueará o envio de qualquer tipo de código malicioso ao servidor.

3 Adulteração de parâmetros (*Parameter tampering*)

Adulteração de parâmetros é um tipo de ataque básico, que pode ser alterados valores de querystrings, cookies e campos de formulários de forma não prevista pelo desenvolvedor e que podem alterar o comportamento da aplicação. Consequentemente podem ocasionar outras falhas de segurança, como por exemplo, o SQL Injection.

Parameter tampering, como dito acima, é uma vulnerabilidade que pode ser explorada simplesmente alterando os campos de formulário que serão enviados ao servidor. Outra forma de alterar os dados que serão enviados ao servidor é utilizar ferramentas que interceptam e registram o tráfego de dados numa rede. Esses softwares podem fornecer informações relevantes sobre o que é enviado e recebido por sua aplicação web. Desta forma podem ser interceptados, alterados e reenviados novamente para o servidor. Este tipo de ataque é conhecido, em inglês, como "*man in the middle attack*". Por isso que a validação do lado do servidor é muito importante e indispensável.

Existem diversas ferramentas e plug-ins que possibilitam a alteração de campos do formulário web - Plugin Toolbar: "Web Developer" para Google Chrome, Firebug, etc.

No ASP.NET Webforms existe uma tag chamada EnableRequestValidation que previne o envio de código maliciosos. Já no ASP.NET MVC pode-se utilizar expressões regulares para limitar ou validar dados. No MVC também se pode utilizar DataAnottations para filtrar informações, por exemplo:

```
[RegularExpression(@"[^a-zA-Z0-9 ]*", ErrorMessage = "Apenas caracteres de
A-Z são permitidos")]
```

Portanto, para evitar a adulteração de dados no lado do cliente:

Garanta que todas as informações provenientes do usuário sejam comparadas ou validadas no lado do servidor.

Nunca confie completamente nas informações enviadas por sua página.

Se você passar algum parâmetro, por exemplo, código de compra, valide se este usuário tem permissão para acessar este código. Por exemplo:

Tabela Pedido (idCliente, idPedido)

Quando buscar um pedido, pesquisar pelos dois id´s, para certificar que é exatamente aquele registro.

Inclua sempre que possível valores de confirmação em seu SQL.

```
select * from pedido where pedidoId=@pedidoId and clienteId=@clienteId
Ou
ctx.Orders.Where(a=>a.OrderId==OrderId && a.CustomerId == customerID);
```

No site da Codeplex (http://mvcsecurity.codeplex.com/) há uma extensão para o Visual Studio que permite validar se houve alteração nos dados do formulário. No Webforms existe o Event Validation, que previne a alteração dos campos hidden.

Esta extensão de segurança para o MVC permite criptografar um campo chave na página utilizando criptografia assimétrica que é usada então para comparar novamente o campo do texto da página para prevenir a adulteração de dados.

Para usar – no método da ação do Controller adicione o atributo e nome da chave:

```
[ValidateAntiModelInjectionAttribute("ClienteId")]
```

E então do lado da View dentro do formulário adicione:

```
@Html.AntiModelInjectionFor(o => o. ClienteId)
```

4 Criptografia e Hashing

A criptografia também pode ser um recurso interessante para aumentar a segurança da sua aplicação. Muitas vezes os dados trafegados pela rede são informações sensíveis e confidenciais que precisam ser preservadas de olhares curiosos. Normalmente as informações pessoais de usuários como senha, login, *tokens* de autenticação, cookies de sessão, cartão de crédito precisam de um cuidado maior. Notícias sobre invasão a banco de dados do sistema acontecem a todo tempo.

A utilização de um protocolo seguro (SSL – *Secure Sockets Layer*) também é recomendável, pois os dados trafegados sob este protocolo são criptografados.

Criptografe apenas se precisar recuperar, caso contrário utilize Hash com Salt.

Criptografia no banco de dados

O SQL Server possui recursos de criptografia para ser executado diretamente no banco de dados. Abaixo, dois tipos de criptografia que podem ser utilizados no banco de dados.

Passphrase: Utiliza-se através do comando EncryptByPassPhrase, passando como parâmetro a sequência de dados(PassPhrase) a ser utilizado para obter a chave de criptografia e o texto a ser convertido, que deve ser armazenado num campo VarBinary. O EncryptByPassPhrase usa o algoritmo DES Triplo para criptografar o texto passado como parâmetro. O processo inverso é realizado através do comando DecryptByPassPhrase.

Exemplo:

Crie uma tabela:

```
Create Table tb_usuarios (nome varchar(100), senha varbinary(100))
```

Insira novo registro com a senha criptografada:

```
INSERT INTO tb_usuarios(nome, senha)VALUES('João', EncryptByPassPhrase
('123','senhaSecreta'))
```

Recupere a senha, descriptografando-a:

```
SELECT nome, convert (varchar (100), DecryptByPassPhrase('123', senha))
from tb_usuarios
```

EncryptByCert: Essa função criptografa dados com a chave pública de um certificado previamente criado. O texto cifrado só poderá ser decifrado com a chave particular correspondente.

Exemplo:

```
CREATE MASTER KEY ENCRYPTION BY
PASSWORD = 'SuperSecretKeyPassword'
GO

CREATE CERTIFICATE MyAppCert
      WITH SUBJECT = 'My App Cert',
      EXPIRY_DATE = '12/21/2012';
GO

Select 'Encrypting Test Text' Texto, EncryptByCert(cert_id('MyAppCert'),
'Texto de Teste ') EncryptedText into #temp

Select * from #temp

Select convert(varchar,DecryptByCert(cert_id('MyAppCert'), EncryptedText))
from #temp

Select convert(varchar,DecryptByCert(cert_id('AnotherAppCert'),
EncryptedText)) from #temp

drop table #temp
```

Criptografia na aplicação

Também há a opção de realizar a criptografia na aplicação. A plataforma .NET contém o algoritmo **Rijndael**, que você deve usar para todos os novos projetos que exigem a criptografia simétrica. Assim você pode usar a classe **RijndaelManaged** para realizar a criptografia e descriptografia simétrica.

Criptografia em arquivos de configuração

Web.config é o arquivo principal de configuração para as aplicações web ASP.NET. Este arquivo é um documento XML que define informações de configurações sobre a aplicação web. O arquivo web.config contém informações que controla o carregamento das páginas, configuração de segurança, o estado da sessão de configuração, aplicação e configurações de compilação. O arquivo Web.config também pode conter informações de conexão com o banco de dados. Se um invasor tentar fazer o download do web.config, por padrão, o ASP.NET não permitirá. Mas há outras formas de conseguir este arquivo.

Para aumentar a segurança sobre este arquivo de configuração é possível utilizar o comando de prompt aspnet_regiis.exe para criptografá-lo.

Exemplo:

```
aspnet_regiis –pef "appSettings" "C:\inetpub\wwwroot\WebApp" –prov
“RsaProtectedConfigurationProvider"
```

O comando acima criptografa o bloco <appSettings> do Web.Config. Quando a aplicação necessitar deste bloco de configuração, o próprio framework fará a descriptografia. Um detalhe, o comando aspnet_regiis.exe deve ser executado no servidor onde a aplicação irá rodar, pois a chave de criptografia padrão do RSA é diferente em cada computador.

Hashing

Hashing, termo em inglês, é a transformação de uma String de caracteres em um valor de tamanho fixo que representa a String original. O *Hashing* é uma função que tem apenas um sentido, ou seja, não há, teoricamente, como convertê-lo novamente ao valor original. Ele é útil para verificar informações como senhas, sessões e assim por diante. Há desvantagem de usar hash na senha, pois a recuperação da senha se torna impossível (neste caso é feito apenas a comparação entre o hash armazenado no banco de dados com o hash da senha que usuário digitou). Os algoritmos de hash mais conhecidos são MD5, SHA-1 e CRC32.

Pode-se usar o MD5 e pensar que está seguro, mas existe uma coisa chamada *Rainbow Tables*, onde um atacante gera uma tabela com o resultado da encriptação contendo todas as palavras de um dicionário, combinando palavras e até adicionando símbolos e dígitos a essas palavras. Através da *Rainbow Table* fica muito fácil (partindo do resultado final da encriptação) descobrir a senha original.

A solução para o problema das *Raibow Tables* é utilizar salts. Salt é uma String complexa que será concatenada a todas e qualquer senha antes de criptografá-la. Funciona como uma chave extra para melhorar o hashing. Dessa forma todas a senhas estarão mais protegida, porém ainda temos um problema: salt fixo.

Todas as senhas serão geradas através do mesmo valor do salt. Esse salt fixo estará salvo em algum lugar do seu sistema. O invasor de posse desse salt, gerará uma nova *Raibow Table* de combinações usando este salt fixo. A solução desse problema será utilizar salts dinâmicos.

Podemos gerar salts dinâmicos para cada senha salva no banco de dados. Salvar ambas, a senha e em outro campo o salt associado a esta senha. Por fim, ainda há um problema. Os algoritmos de encriptação (MD5 e SHA1) são criados para serem extremamente rápidos, pois são usados na validação e verificação de integridade de arquivos. Quanto mais rápido o algoritmos, mais fácil se tornam os ataques por força bruta (com ou sem *Raibow Tables*).

Uma solução para aumentar ainda mais a segurança da criptografia, é atrasar o algoritmo. Por exemplo:

```
// Encripta o hash 100 vezes
for (i = 0; i < 100; i++)
      hash = md5(hash);
```

Ou seja, o ataque por força bruta irá demorar pelo menos cem vezes mais tempo. Este processo de múltiplas iterações é mais lento, porém é mais seguro.

5 Cross-Site Request Forgery (CSRF - Falsificação de pedido)

CSRF é um tipo de ataque que força o usuário a executar uma ação não desejada numa aplicação web que ele está autenticado. Com uma pequena ajuda da engenharia social (como enviar um link via e-mail), um atacante pode forçar o usuário de uma aplicação web executar ações que o atacante desejar. Um ataque de CSRF bem sucedido pode comprometer os dados do usuário. Caso este usuário for um administrador, pode-se comprometer a aplicação inteira.

Algumas pessoas podem achar que CSRF e XSS são as mesmas coisas, porém cada um tem uma característica específica. Na vulnerabilidade do XSS o usuário confia na integridade do site, que pode ter algum script injetado anteriormente por um atacante. Já na vulnerabilidade do CSRF, o site confia nas

requisições do usuário, e este pode ser enganado por outro site que envia requisições não autorizadas para o site confiável.

CSRF é um ataque que engana a vítima através do carregamento de uma página que contém um pedido malicioso. Ele é mal intencionado no sentido de que ele herda a identidade e os privilégios da vítima para executar uma função indesejada em nome da vítima, como a mudança de endereço de e-mail da vítima, endereço residencial, senha ou até mesmo comprar algo no lugar dela. Desta forma, o atacante pode fazer a vítima executar ações que não tinha a intenção.

Há inúmeras maneiras de enganar o usuário e carregar informações não desejadas numa aplicação web. A fim de executar um ataque, primeiro devemos entender como gerar um pedido malicioso para a vítima executar. Vamos considerar o seguinte exemplo.

João deseja transferir R$500 a Maria. O pedido gerado por João será semelhante a isso:

POST http://banco.com/transferir.do HTTP/1.1
...
Content-Length: 21;
conta=Maria&valor=500

Entretanto, João percebe que a mesma aplicação web irá executar a mesma transferência usando parâmetros através da URL:

GET http://banco.com/transferir.do?conta=Maria&valor=500 HTTP/1.1

João então decide explorar esta vulnerabilidade submetendo outros valores:

GET http://banco.com/transferir.do?conta=Joao&valor=50000 HTTP/1.1

Agora João vai tentar enganar Maria, para que ela submeta a requisição. O método mais básico é enviar para Maria um e-mail contendo o HTML:

```
<a href="http://banco.com/ transferir.do?
transferir.do?conta=Joao&valor=50000>Veja minhas fotos!</a>
```

Caso Maria esteja autenticada na aplicação web, quando ela clicar no link, neste caso hipotético, ela irá transferir R$50.000,00 para João. Ele também pode enviar uma imagem falsa, que irá executar o mesmo comando:

```
<img src="http://banco.com/ transferir.do?
transferir.do?conta=Joao&valor=50000 with=1 height=1 border=0>
```

Se esta tag de imagem for incluída no e-mail, Maria apenas observará uma pequena caixa indicando que o navegador não pode carregar a imagem. Neste caso, o navegador continuará submetendo requisições para o banco.com sem nenhuma indicação visual indicando que a transferência foi realizada.

CSRF é uma vulnerabilidade pouco conhecida pelos desenvolvedores, porém existe em muitos sites. Há muitos sites conhecidos que ainda permite este tipo de ataque. Há muitos tipos de proteções para este tipo de vulnerabilidade. A solução mais efetiva é utilizar um gerador randômico de *token*.

A prevenção do CSRF no ASP.NET MVC é simples. O framework do ASP.NET MVC inclui uma coleção de Helpers[4], entre eles o Html.AntiForgeryToken(), que permite detectar e bloquear ataques do tipo CSRF através de tokens específicos para cada usuário.

Exemplo de como prevenir o CSRF:

Para utilizar este *Html Helper* para proteger o seu formulário, insira o código abaixo:

```
<% using(Html.Form("CadastroUsuario", "SubmitUpdate")) { %>
<%=Html.AntiForgeryToken() %>
<!-- restante do formulário vem aqui -->
<% } %>
```

O HTML gerado será:

```
<form action="/UserProfile/SubmitUpdate" method="post">    <input
name="__RequestVerificationToken" type="hidden"
value="saTFWpkKN0BYazFtN6c4YbZAmsEwG0srqlUqqloi/fVgeV2ciIFVmelvzwRZpArs" />
       <!-- restante do formulário vem aqui -->
       </form>
```

Html.AntiForgeryToken() irá gerar um cookie chamado __RequestVerificationToken com o mesmo nome do campo hidden.

O próximo passo para validar os dados submetidos é adicionar o filtro [ValidateAntiForgeryToken] no método do controller. Por exemplo:

```
[ValidateAntiForgeryToken]
public ViewResult SubmitUpdate()
{
    // ... etc
}
```

Dessa forma a requisição terá um cookie e um campo hidden de validação que irá prevenir o CSRF. Se o atacante não possuir este token, ele não poderá forjar o envio de um formulário válido. Dessa forma os usuários autenticados não serão incomodados, e este mecanismo funciona de forma totalmente transparente.

Conclusão

[4] Html Helpers são métodos que permitem exibir no navegador elementos HTML de forma mais ágil e simples.

A segurança sempre foi classificada como tema importante, porém é pouco adotado pelos desenvolvedores. Atualmente, é muito comum encontrar falhas de segurança em diversos sistemas. Por esse motivo a pesquisa e o estudo sobre as tecnologias de desenvolvimento e suas “brechas” de segurança são relevantes.

Este artigo apresentou alguns conceitos e informações sobre os principais tipos de vulnerabilidades mais exploradas. Cada um dos itens comentados neste artigo poderia produzir outros diversos artigos separadamente. Porém este artigo relacionou as principais vulnerabilidades e indicou soluções de como produzir soluções mais seguras. Mostrou-se que as soluções para os ataques mais comuns são simples. A criptografia é um recurso que deve ser utilizado pra preservar dados sensíveis na sua aplicação. Tanto de dados como senha e cartões de crédito, como de protocolo de comunicação entre servidor e cliente – através do SSL.

O SQL Injection e o XSS são vulnerabilidades bastante conhecidas e facilmente evitadas. Já adulteração de valores exige maior atenção do desenvolvedor durante a codificação, pois poderá permitir que dados falsificados sejam inseridos na aplicação. A criptografia e hashing foram inseridos neste artigo como recursos que aumentam a segurança, no sentido que podem proteger os dados sensíveis da aplicação, tais como senha, cartões de crédito, etc. Por fim, o CSRF é uma falha de segurança que explora uma sessão válida e autenticada pelo usuário num determinado navegador. Em conjunto com a engenharia social pode-se facilmente ser explorada tal falha.

As vulnerabilidades apresentadas neste artigo também podem ser exploradas em conjunto. O CSRF pode ser explorado dentro de uma falha de segurança do XSS. O SQL Injection, apesar de ser uma das vulnerabilidades mais conhecidas, ainda é muito comum encontrar. Observa-se também que a adulteração de informações e principalmente o CSRF vêm sendo muito utilizadas atualmente.

Apesar de existirem essas e diversas outras vulnerabilidades nas aplicações web, o framework .NET possui recursos que permitem que elas sejam facilmente corrigidas.

Trabalho futuro

Como trabalho futuro, são linhas de estudos os seguintes itens: i) detalhar cada tipo de falha, pois existem derivações de ataques para cada tipo de

vulnerabilidade; ii) mapear e levantar dados estatísticos do tipo e a frequência com que ocorrem os ataques; iii) desenvolver ferramenta automatizada que detecte as vulnerabilidades de uma aplicação *web* – emitindo relatório com prioridade e grau do risco; iv) desenvolver aplicativo de monitoramento de requisições falsas ao servidor; v) estudar outras vulnerabilidades e falhas de segurança para plataforma .NET.

Referências

Dorrans, Barry. **Beginning ASP.NET Security** (Wrox Programmer to Programmer). Wrox, 2010.

LAMBERT, N. **Use of Query tokenization to detect and prevent SQL injection attacks**. 09 julho 2010. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

SHAR, Lwin Khin. **Auditing the defense against cross site scripting in web applications**. 26 julho 2010. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

SHAR, Lwin Khin. **Defending against Cross-Site Scripting Attacks**. Março 2012. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

RABIN, M.O. **Hashing on strings, cryptography, and protection of privacy**. 11 junho 2007. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

ALEXENKO, T. **Cross-Site Request Forgery: Attack and Defense**. 09 Janeiro 2010. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

SIDDIQUI, M.S. **Cross site request forgery: A common web application weakness**. 27 maio 2011. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

SHAHRIAR, H. **Client-Side Detection of Cross-Site Request Forgery Attacks**. 01 novembro 2010. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

SINGH, M. **Choosing Best Hashing Strategies and Hash Functions**. 06 março 2009. Disponível em: <http://ieeexplore.ieee.org>. Acesso em: 10/10/2012

MVC Security Extensions. Disponível em: <http://mvcsecurity.codeplex.com>. Acesso em: 01/09/2012

Secure Coding. Disponível em: <http://secure-coding.com/>. Acesso em: 01/09/2012

The Open Web Application Security Project. Disponível em: <https://www.owasp.org>. Acesso em: 02/09/2012

EncryptByPassPhrase (Transact-SQL). Disponível em: <http://msdn.microsoft.com/en-us/library/ms190357(v=sql.90).aspx>. Acesso em: 02/09/2012

HtmlHelper.AntiForgeryToken Method. Disponível em: <http://msdn.microsoft.com/en-us/library/dd470175(v=vs.108).aspx>. Acesso em: 02/09/2012

AFONSO,VITOR MONTE. **Um sistema para análise e detecção de ataques ao navegador *Web.*** 06 de outubro 2011. <UNICAMP>